Notes for the course advanced algorithms
January 2000

Johan Håstad
email:johanh@nada.kth.se

Contents

Chapter 1

Introduction

The aim of the course "Advanced Algorithms" is at least twofold. One aim is to describe a couple of the classical algorithms which are not taught in a first algorithms course. The second is to give a general understanding of efficient algorithms and to give the student a better understanding for how to design and analyze efficient algorithms. The overall approach of the course is theoretical, we work with pencil and paper. The main emphasis of the course is to get an algorithm with a good asymptotic running time. In most cases this coincides with efficient in practice and the most notable example where this is not true is matrix multiplication where the implied constants are too large to make the algorithms be of practical value.

Although the lectures are theoretical, students are asked to implement some basic algorithms in the homework sets. Our hope is that this gives at least some balance between practice and theory.

The current set of lecture notes include all topics covered in the course in its past four appearances in spring 1995, spring 1997, fall 1997, and fall 1998. The course in 1995 was mostly taken by graduate student while the later courses were mostly taken by undergraduate students. Thus some of the more advanced topics, like matrix multiplication, lattice basis reduction and provable polynomial time for integer polynomial factorization, was only covered in 1995 and may not be covered in coming years. However, for the interested reader we have kept those sections in these notes. Also when it comes to the rest of the material we do not expect to cover all algorithms each time the course is given. The choice of which algorithms to cover is done at the beginning of a course to make it possible for the participants to influence this decision. The choice is not limited to the algorithms included in these notes and I would be happy to include any desired new topic that would be on a topic agreeing with the course philosophy.

The present set of notes is partly based on lecture notes taken by participants in earlier courses. Many things to Lars Arvestad, Ann Bengtsson, Christer Berg, Marcus Better, Johnny Bigert, Lars Engebretsen, Mikael Goldmann, Daniel Hegner, Anders Holst, Peter Karpinski, Marcus Lagergren, Mikael Larsson, Christer Liedholm, Per Lindberger, Joakim Meldahl, Mats Näslund, Henrik Ståhl, Claes Thornberg, Anders Törlind, Staffan Ulfberg, Bengt Werner, and Douglas Wikstrm for taking such notes. I am also indebted to later students as well as Kousha Etessami for pointing out errors. Of such later students Anna

Redz deserves special credit for many comments on essentially all the chapters. Special thanks also to Staffan Ulfberg for completely rewriting the chapter on quantum computation. Of course, I must myself take full responsibility for any errors remaining in the notes. I hope to update these lecture notes every year to decrease the number of misprints and to clarify bad explanations. Any new topics covered will also be added to the notes.

We ask the reader to treat these notes for what they are, namely lecture notes. The standard of the text is likely to remain uneven (but will hopefully improve over time) and this is in particular true for the academic strictness. We try our best to be correct everywhere, but one definite shortcoming is given by the references. For some topics accurate references are given while for others the material is presented without due credit. We apologize for this but our aim is to improve the quality also in this respect and we feel that an uneven standard is better than essentially no references anywhere.

Chapter 2

Notation and basics

Most of the time we will not worry too much about our model of computation and one should best think of a machine that can do "standard operations" (like arithmetic operations and comparisons) on computer words of "reasonable size". In theory, we allow $O(\log n)$ size words and in practice most of our algorithms will only need arithmetic on 64 bit words. We say that an algorithm runs in time $O(f(n))$ when, for some absolute constant c, it uses at most $cf(n)$ such standard operations on inputs of length n. We say that an algorithm runs in time $\Omega(f(n))$ if, for some absolute constant c, for infinitely many n there is some input for which it does $cf(n)$ operations.

2.1 A couple of basic algorithms

Before we enter the discussion of more advanced algorithms let us remind the reader of two basic algorithms. Computing the greatest common divisor by the Euclidean algorithm and solving systems of linear equations by Gaussian elimination.

2.1.1 Greatest common divisor

The greatest common divisor of two numbers a and b is usually denoted by $gcd(a, b)$ and is defined to be the largest integer that divides both a and b. The most famous algorithm for computing $gcd(a, b)$ is the Euclidean algorithm and it proceeds as follows. For simplicity assume that $a > b \geq 0$.

Euclid(a, b) =
    While $b \neq 0$ **do**
        $d := \lfloor a/b \rfloor$
        $tmp = b$
        $b = a - b * d$
        $a = tmp$
    od
    Return a

Here, $\lfloor a/b \rfloor$ denotes the largest integer not greater than a/b.

Example 2.1. *Consider the following example computing gcd$(518, 721)$ where we display the numbers a and b that appear during the calculation without keeping track of which number is which*

$$\begin{aligned}
721 &= 518 + 203 \\
518 &= 2 \cdot 203 + 112 \\
203 &= 112 + 91 \\
112 &= 91 + 21 \\
91 &= 4 \cdot 21 + 7 \\
21 &= 3 \cdot 7
\end{aligned}$$

and thus gcd$(518, 721) = 7$. In some situations, more information that just the greatest common divisor is needed and an important extension is the extended Euclidean algorithm *which apart from computing gcd(a, b) also finds two integers x and y such that $ax + by =$ gcd(a, b). We describe this algorithm simply by running it in the above example.*

$$\begin{aligned}
7 &= 91 - 4 \cdot 21 = 91 - 4 \cdot (112 - 91) = \\
&= 5 \cdot 91 - 4 \cdot 112 = 5 \cdot (203 - 112) - 4 \cdot 112 = \\
&= 5 \cdot 203 - 9 \cdot 112 = 5 \cdot 203 - 9 \cdot (518 - 2 \cdot 203) = \\
&= 23 \cdot 203 - 9 \cdot 518 = 23 \cdot (721 - 518) - 9 \cdot 588 = \\
&= 23 \cdot 721 - 32 \cdot 518
\end{aligned}$$

The efficiency of the Euclidean depends on the implementation, but even the most naive implementation is rather efficient. If a and b are n-bit numbers it is hot hard to prove that the algorithm terminates after $O(n)$ iterations. Furthermore, the cost of one iteration is dominated by the cost of the division and the multiplication. Both these operations can be done in time $O(n^2)$ and thus the total running time of the algorithm is bounded by $O(n^3)$. This analysis also applies to the extended Euclidean algorithm and we state this as a theorem.

Theorem 2.2. *The extended Euclidean algorithm applied to n-bit integers and implemented in a naive way runs in time $O(n^3)$.*

Apart from computations over integers we are also be interested in doing the same operations on polynomials with coefficients in different domains. In this situation, the algorithm is, in one sense, even simpler. We can simply choose the multiplier d to be a suitable monomial which decreases the degree of one of the polynomials by at least one. It is then easy to see that the number of iterations is $O(n)$ (as before) and moreover that each iteration can be implemented in $O(n)$ operations over the domain in question. Thus the estimate for the running time can, in this case, be improved to $O(n^2)$ operations. We state this as a theorem.

Theorem 2.3. *The extended Euclidean algorithm applied to polynomials of degree n over any field runs in $O(n^2)$ field operations.*

It is not hard to improve also the running time of the Euclidean algorithm over the integers, but we ignore this since it just complicates the description and this improvement is not essential for us.

2.1.2 Systems of linear equations

We assume that the reader is familiar with the method of Gaussian elimination for solving systems of linear equations. The method is usually presented for equations over rational numbers but the method works equally well over finite fields (if you do not know what these are, just think of the integers modulo p where p is a prime). We illustrate this by solving the following system of equations:

$$\begin{cases} x_1+ & x_2+ & x_3+ & x_4 & = 1 \\ x_1+ & & x_3 & & = 0 \\ & x_2+ & x_3+ & x_4 & = 0 \\ x_1+ & x_2+ & & & = 1 \end{cases}$$

over the finite field of two elements, (*i. e.* the integers mod 2) where variables take values 0 or 1 and addition is defined as exclusive-or. We add the first equation to equations 2 and 4 obtaining:

$$\begin{cases} x_1+ & x_2+ & x_3+ & x_4 & = 1 \\ & x_2+ & & x_4 & = 1 \\ & x_2+ & x_3+ & x_4 & = 0 \\ & & x_3+ & x_4 & = 0 \end{cases}$$

We proceed by adding equation 2 to equations 1 and 3 obtaining:

$$\begin{cases} x_1+ & & x_3 & & = 0 \\ & x_2+ & & x_4 & = 1 \\ & & x_3 & & = 1 \\ & & x_3+ & x_4 & = 0 \end{cases}$$

Adding equation 3 to equations 1 and 4 gives:

$$\begin{cases} x_1+ & & & & = 1 \\ & x_2+ & & x_4 & = 1 \\ & & x_3 & & = 1 \\ & & & x_4 & = 1 \end{cases}$$

and finally adding equation 4 to equation 2 gives the complete solution $(x_1, x_2, x_3, x_4) = (1, 0, 1, 1)$. For a system of n equations in n unknowns we have n iterations in which we add a single equation to at most n other equations. Since each addition of two equations can be done in time $O(n)$ the overall running time is $O(n^3)$. For future reference we state this as theorem:

Theorem 2.4. *A system of linear equations in n unknowns over a finite field can be solved in time $O(n^3)$ field operations by Gaussian elimination.*

Note that in practice the constant in front of n^3 in the running time of this algorithm is very small. In actual implementations we can store the coefficients as bit-vectors in computer words and thus individual machine instructions can perform what is counted as 32 (or 64 if the word-size is 64) operations above.

2.1.3 Depth first search

A basic algorithm for searching a graph is depth-first search. Since it is many times only described for trees we describe it here for general graphs. We have a graph with nodes V and edges E. It is best described as a recursive algorithm. Before the algorithm starts each $v \in V$ is marked as 'not visited'. $DFS(v)$ is a recursive procedure given below.

$$DFS(v)$$

```
For each (v, w) ∈ E do
    if w not visited do
        mark w visited
        DFS(w)
    od
od
Return
```

This algorithm can be used as a subroutine for many tasks. We state the most simple property of the algorithm.

Theorem 2.5. *When started at vertex v, $DFS(v)$ marks exactly the nodes in the connected component of v as 'visited' and then halts. If the graph has n nodes and m edges, it runs in time $O(n + m)$.*

Chapter 3

Primality testing

Given an integer, N, how do we find out whether it is prime? We should think of N as a large integer with at least 100 decimal digits. Assume that N has n bits, that is, $N \approx 2^n$ and that we are interested in algorithms that run in time polynomial in n.

The reason for studying this problem is twofold. Firstly, it is a fundamental problem of computer science and mathematics. Secondly, it has applications in cryptography through the RSA cryptosystem. For details of the latter see [13].

The naive approach to determine if N is prime is to try to divide N by all numbers smaller than $\sqrt{N}$. Since there are $\sqrt{N} \approx \sqrt{2^n}$ numbers to try, this takes time $\Omega(2^{n/2})$, which is infeasible if n is large.

A better idea is to make use of Fermat's theorem which says that if p is prime then $a^{p-1} \equiv 1 \bmod p$ for all a such that $1 \leq a \leq p-1$. However, before we use this theorem, there are two problems that we have to consider:

1. Is this method efficient?

2. Is the converse of Fermat's theorem true; does $a^{N-1} \equiv 1 \bmod N$ imply that N is prime?

To begin with the first problem, the question is how many operations do we need in order to compute $a^{N-1} \bmod N$ where $N \approx 2^n$.

The number a^{N-1} is enourmous containing at least N digits and thus we cannot compute this number explicitly. But since we are only interested in the final answer modulo N, we need only remember partial results modulo N (think about this fact for a second). This implies that $O(n)$ bits are sufficient for each number computed.

If we simply keep multiplying by a, computing a, a^2, ... a^{N-1}, we do $N-2$ multiplications and thus something more efficient is needed. Squaring a number on the form a^i gives a^{2i} and thus we can have exponents that grow exponentially fast in the number of operations. Squaring and multiplying by a turn out to be the only operations needed and the general technique, which is called repeated squaring, is probably best illustrated by an example.

Example 3.1. *Suppose we want to compute* $2^{154} \bmod 155$. *Writing* 154 *in binary we get* 10011010. *The powers to which we compute* 2 *are then, written in binary 1, 10, 100, 1001, 10011, 100110, 1001101, and 10011010 or written* *in decimal 1,2,4,9,19,38, 77, and 154. It is done as follows.*

$$\begin{aligned}
2^1 &\equiv 2 \cdot 1^2 \equiv 2 \bmod 155\\
2^2 &\equiv 2^2 \equiv 4 \bmod 155\\
2^4 &\equiv 4^2 \equiv 16 \bmod 155\\
2^9 &\equiv 2 \cdot 16^2 \equiv 47 \bmod 155\\
2^{19} &\equiv 2 \cdot 47^2 \equiv 78 \bmod 155\\
2^{38} &\equiv 78^2 \equiv 39 \bmod 155\\
2^{77} &\equiv 2 \cdot 39^2 \equiv 97 \bmod 155\\
2^{154} &\equiv 97^2 \equiv 109 \bmod 155.
\end{aligned}$$

From this we can also conclude that 155 is not prime since if it was the answer would have been 1.

Since N has n binary digits, $a^{N-1} \bmod N$ can be computed with $2n$ multiplications of n-digit numbers together with n modular reductions. Since one multiplication and one modular reduction of n-bit numbers can easily be done in $O(n^2)$ operations on numbers of bounded size, the total time for modular exponentiation is $O(n^3)$. Hence, the answer to the first question, whether Fermat's theorem is efficient, is 'yes'.

We turn to question of whether the condition that $a^{N-1} \equiv 1 \bmod N$ implies that N is prime. If this condition is true for all a between 1 and $p-1$ then N must be prime since $\gcd(a, N) = 1$ for $1 \le a \le N-1$ which clearly implies that N is prime. However, checking all these values for a is clearly inefficient and thus we would like a much stronger converse. In the best of all worlds we would have hoped that $a^{N-1} \equiv 1 \bmod N$ for a small set of a's implies that N is prime. This is, unfortunately, not true and even requiring it to be true for all a that are relatively prime to N is not sufficient. The counterexamples are given by the so called *Carmichael numbers*, the smallest of which is 561. From this it follows that we cannot use Fermat's theorem as it is to separate primes from composites, but we shall see that it is possible to change it slightly so that it, at least for most a's, reveals composites – even Carmichael numbers. Before we continue we need to recall the Chinese remainder theorem.

3.1 Chinese remainder theorem

Let $N = p_1 p_2 \cdots p_r$ where p_i are relatively prime, *i. e.* , $gcd(p_i, p_j) = 1$ for $i \neq j$. Note, in particular that the latter assumption is true when the p_i are different primes. The *Chinese remainder theorem* states that the equation

$$\begin{aligned}
x &\equiv a_1 \bmod p_1,\\
x &\equiv a_2 \bmod p_2,\\
&\vdots\\
x &\equiv a_r \bmod p_r,
\end{aligned}$$

has a unique solution mod$N$.

Example 3.2. *Let* $N = 15 = 3 \cdot 5$. *Then each number* $a \bmod 15$ *can be represented as* $(a \bmod 3, a \bmod 5)$, *as follows.*

$$\begin{array}{lll} 0 \sim (0,0), & 5 \sim (2,0), & 10 \sim (1,0), \\ 1 \sim (1,1), & 6 \sim (0,1), & 11 \sim (2,1), \\ 2 \sim (2,2), & 7 \sim (1,2), & 12 \sim (0,2), \\ 3 \sim (0,3), & 8 \sim (2,3), & 13 \sim (1,3), \\ 4 \sim (1,4), & 9 \sim (0,4), & 14 \sim (2,4). \end{array}$$

When computing, for example, $8 \cdot 14$ *we can use this representation. Working component-wise we get*

$$8 \cdot 14 \sim (2,3) \cdot (2,4) = (2 \cdot 2, 3 \cdot 4) = (4,12) \equiv (1,2) \sim 7.$$

Since $8 \cdot 14 = 112 \equiv 7 \bmod 15$ *this is correct and it is not difficult to convince oneself that this is a correct procedure in general.*

From the Chinese remainder theorem it follows that choosing x randomly in $\{0, 1 \ldots N-1\}$ where $N = \prod_{i=1}^{r} p_i$ is be equivalent to, independently for different i, choosing a_i randomly in $\{0, 1, \ldots p_i - 1\}$ and then determining x by requiring $x \equiv a_i \mod p_i$ for $i = 1, 2 \ldots r$.

3.1.1 Chinese remainder theorem in practice

It is sometimes necessary to go back and forth between the representations $x \bmod N$ and $(a_1, a_2 \ldots a_r)$ where $x \equiv a_i \bmod p_i$. Going from x to the other representation is easy but going the other direction requires some thought. One could try all values of x but this is inefficient, hence some other method is needed. We show how to do this for $r = 2$. The generalization to more factors follows from applying the argument repeatedly.

Thus we have

$$\begin{cases} x \equiv a_1 \bmod p_1 \\ x \equiv a_2 \bmod p_2 \end{cases}$$

and we want to find $x \bmod p_1 p_2$. It turns out that it is sufficient to solve this for $(a_1, a_2) = (1, 0)$ and $(a_1, a_2) = (0, 1)$. Namely, suppose

$$\begin{cases} U_1 \equiv 1 \bmod p_1 \\ U_1 \equiv 0 \bmod p_2 \end{cases}$$

and

$$\begin{cases} U_2 \equiv 0 \bmod p_1 \\ U_2 \equiv 1 \bmod p_2 \end{cases}$$

Then it is not hard to see that

$$x \equiv a_1 U_1 + a_2 U_2 \bmod p_1 p_2,$$

fulfills the first set the equations above.

One question remains; how do we find U_1 and U_2? Euclid's extended algorithm on (p_1, p_2) gives $1 = ap_1 + bp_2$, and we identify $U_1 = bp_2$ and $U_2 = ap_1$. It is not difficult to extend this to larger r (we leave the details as an exercise) and we state the results as a theorem

Theorem 3.3. *(Efficient Chinese remainder theorem) Let* $N = \prod_{i=1}^{r} p_i$ *be an* n *bit integer where* p_i *are pairwise relatively prime. Then for any* $(a_i)_{i=1}^{r}$ *there is a unique* x *satisfying* $x \equiv a_i \bmod p_i$ *and this* x *can be found in time* $O(n^3)$.

Example 3.4. *Let's say we have the following equations:*

$$\begin{aligned} x &\equiv 4 \bmod 23 \\ x &\equiv 7 \bmod 15. \end{aligned}$$

Euclid's extended algorithm gives us

$$1 = 2 \cdot 23 - 3 \cdot 15.$$

This gives

$$\begin{aligned} U_1 &= -45 \\ U_2 &= 46 \end{aligned}$$

and that

$$x = 4 \cdot (-45) + 7 \cdot 46 \bmod 23 \cdot 15 \equiv 142 \bmod 345.$$

As with the Euclidean algorithm let us remark that everything works equally well when considering a polynomial $N(x)$ which is product of relatively prime polynomials. We omit the details.

3.2 The Miller-Rabin primality test

Let us return to primality and start by describing the new test we will analyze. It is traditionally called the Miller-Rabin primality test. Gary Miller originally came up with the test and Michael Rabin realized that the proper setting was that of a probabilistic algorithm. Let $N-1 = 2^s t$ where t is odd and $s \geq 1$ since it easy to determine whether an even number is a prime[1]. Pick a randomly, then compute

1. $u_0 \equiv a^t \bmod N$
2. $u_{i+1} \equiv u_i^2 \bmod N$, for $i = 1, 2, \ldots s$.
3. Accept, *i. e.* , declare "probably prime" if u_0 is 1 or some u_i, $0 \leq i < s$ equals -1.

Note that $u_s \equiv a^{2^s t} \equiv a^{N-1}$ and thus it should take the value 1 modulo N if N is a prime. Note that this condition is implied by the acceptance criteria, since if $u_i \equiv -1$ then $u_j \equiv 1$ for $i+1 \leq j \leq s$. In a more general test we repeat the procedure k times with independent values of a and declare N to be "probably prime" if the result is probably prime each time.

Assume that N is prime. Then we claim we always declare N as "probably prime". This follows since we know that $u_s \equiv 1 \bmod N$, so either we have that $u_0 \equiv 1 \bmod N$ or else there is a last u_i such that $u_i \not\equiv 1 \bmod N$ and $u_{i+1} \equiv u_i^2 \equiv 1 \bmod N$. But if N is prime, then the equation $x^2 \equiv 1 \bmod N$ has only the solutions $x \equiv \pm 1$ and so we must have $u_i \equiv -1 \bmod N$ which means that N is declared as "probably prime".

On the other hand, if N is not prime, we want to prove that half of the a's reveal that N is not prime. The intuition to why this is the case can be

[1] An even number is prime iff it is 2

described as follows. Suppose that $N = p_1 p_2 \cdots p_r$ and that we pick an $a \bmod N$ at random. By the Chinese remainder theorem picking a random $a \bmod N$ is just independently picking a random $a_1 \bmod p_1$, a random $a_2 \bmod p_2$, and so on. Now, in order for $u_i \bmod N$ to equal -1 we must, by the Chinese remainder theorem, have that

$$\begin{aligned} u_i &\equiv -1 \bmod p_1 \\ u_i &\equiv -1 \bmod p_2 \\ &\vdots \\ u_i &\equiv -1 \bmod p_r. \end{aligned}$$

When we view the computation modulo N as taking place modulo p_i for the different i we can suspect that this situation is unlikely to occur.

Example 3.5. *Suppose $N = 73$, giving $s = 3$ and $t = 9$, and $a = 3$. Then $u_0 = 46$, $u_1 = 72 = -1$, $u_2 = u_3 = 1$ and the test accepts which it should since 73 is prime.*

Example 3.6. *A more interesting example is when N is the Carmichael number $561 = 3 \cdot 11 \cdot 17$ and a is equal to 2. In this case $561 - 1 = 16 \cdot 35$ so that $s = 4$ and $t = 35$. Since $2^{35} \equiv 263 \bmod 561$ we have the following table.*

	mod3	mod11	mod17	mod561
u_0	-1	-1	8	263
u_1	1	1	13	166
u_2	1	1	-1	67
u_3	1	1	1	1
u_4	1	1	1	1

Here, we have that no row consists of only -1 modulo the three primes in 561. Hence, we never get $-1 \bmod 561$ and since also $u_0 \not\equiv 1 \bmod 561$ the Miller-Rabin test correctly says that 561 is composite.

The essential property of the algorithm is now the following.

Theorem 3.7. *The Miller-Rabin primality test repeated k times always declares a prime as "probable prime". A composite number is declared "probably prime" only with probability at most 2^{-k}. The algorithm runs in time $O(kn^3)$.*

Proof. The correctness when N is prime was established above. The bound on the running time is achieved by the repeated squaring procedure described earlier. The claim for composite number follows from the claim below. □

Claim 3.8. *If N is composite, the probability that the Miller-Rabin algorithm outputs "probably prime" in any single iteration is bounded by 1/2.*

Before proving the second part, let us note that if we for instance choose $k = 100$, the algorithm answers correctly with an overwhelming probability: $1 - 2^{-100}$. This probability is small compared to other error sources such as error is the program, error in the compiler, error in the operating system or error in the hardware. Thus it is of small practical significance.

To prove Claim 3.8 we distinguish two cases:
1. N is a prime power, $N = p^b$, $b \geq 2$ and
2. N has at least two distinct prime factors.

Proof. (In case 1.) In this case N will not even pass the criterion in Fermat's theorem, *i. e.* for most a we have $a^{N-1} \not\equiv 1 \bmod N$. Assume $N = p^b$, $b \geq 2$. If indeed, $a^{N-1} \equiv 1 \bmod N$, we find many a_i such that $a_i^{N-1} \not\equiv 1 \bmod N$. This can be done by noting that if $a^{N-1} \equiv 1 \bmod N$ and we set $a_i = (a + ip)$ where $1 \leq i < p$, then:

$$\begin{aligned} a_i^{N-1} &\equiv (a+ip)^{N-1} \equiv \sum_{j=0}^{N-1} \binom{N-1}{j} a^{N-1-j}(ip)^j \\ &\equiv a^{N-1} + (N-1)ipa^{N-2} + p^2 \sum_{j=2}^{N-1} \binom{N-1}{j} a^j (ip)^{j-2} \\ &\equiv a^{N-1} + (N-1)ipa^{N-2} \\ &\equiv 1 + (N-1)ipa^{N-2} \bmod p^2. \end{aligned}$$

But $(N-1)ipa^{N-2} \not\equiv 0 \bmod p^2$ since $p \nmid N-1$, $p \nmid i$ and $p \nmid a^{N-2}$. We have thus for any a that passes the criterion identified $p-1$ other a which do not pass the criterion. Furthermore, different passing a give rise to different non-passing numbers (this is not completely obvious so please verify this). We can conclude that the test accept with probability at most $1/p$. $\square$

To prove Claim 3.8 in case two, *i. e.* when N is not a prime power, we need the following two lemmas.

Lemma 3.9. *Let p be a prime where $p - 1 = 2^{r'}t'$ where t' is odd. Then*

$$\Pr_{a \in \mathbb{Z}_p^*}[a^{2^r t} \equiv -1 \mod p \mid a^{2^s t} \equiv 1 \mod p]$$

where t is odd is bounded by $\max(2^{r-r'}, 2^{r-s})$ *when* $r < \min(r', s)$ *and 0 otherwise.*

Lemma 3.10. *Let p be a prime where $p - 1 = 2^{r'}t'$ where t' is odd. Suppose $s < r'$ then*

$$\Pr_{a \in \mathbb{Z}_p^*}[a^{2^s t} \equiv 1 \mod p]$$

where t is odd is bounded by $2^{s-r'}$.

Proof. (Of Lemma 3.9 and 3.10.) We leave this as an exercise. As a hint, you might consider the fact that the multiplicative group mod p is isomorphic to the additive group mod $p - 1$. In this isomorphism 1 corresponds to 0 and -1 to $(p - 1)/2$. Squaring in the multiplicative setting corresponds to multiplication by 2 in the additive setting and you can then analyze what power of two divides the numbers in question.

The following intuition (which can be proved) might be of help: If $x^{2l} \equiv 1 \bmod p$, what was $x^l \bmod p$? The only roots of $1 \bmod p$ are ± 1. It seems natural that half of the roots are 1 and half -1 and this is also the case under most circumstances. $\square$

We now proceed with the proof of Claim 3.8.

Proof. (In case 2.) Here assume $N = p_1 p_2 M$ for some $M \in \mathbb{Z}$. As before, let $N - 1 = 2^s t$, t odd. For N to pass the test as "probably prime", we must have $u_0 \equiv 1 \bmod N$ *or* $u_r \equiv -1 \bmod N$ for some $r < s$. These events are mutually exclusive.

(i) If $u_0 \equiv 1 \bmod N$, now by the Chinese remainder theorem, $u_0 \equiv 1 \bmod p_1$ *and* $u_0 \equiv 1 \bmod p_2$.

(ii) If $u_r \equiv -1 \bmod N$ for some $r < s$, by the same reasoning, $u_r \equiv -1 \bmod p_1$ and $u_r \equiv -1 \bmod p_2$ which also implies $u_s \equiv 1 \bmod p_1$, $u_s \equiv 1 \bmod p_2$.

The error probability is at most the sum of the probabilities of the two events (i) and (ii) above. Let us analyze (ii) first.

If we chose a random $a \bmod N$, then we also choose $a \bmod p_1$, $a \bmod p_2$ uniformly and independently. Assume $p_1 - 1 = 2^{r_1} t_1$ and $p_2 - 1 = 2^{r_2} t_2$ with t_1, t_2 odd. Without loss of generality, assume $r_1 \geq r_2$. We distinguish two separate cases:

(a) $s < r_1$. Then, by Lemma 3.10,

$$\Pr_{a \in \mathbb{Z}_{p_1}^*} [a^{2^s t} \equiv 1 \bmod p_1] \leq 2^{s - r_1} \leq \frac{1}{2} \tag{3.1}$$

and we are clearly done since this also covers case (i).

(b) $s \geq r_1$. Suppose $a^{2^s t} \equiv 1$ both $\bmod\ p_1$ *and* $\bmod\ p_2$. Then, by Lemma 3.9, the probability of having $a^{2^r t} \equiv -1 \bmod p_1$ *and* $a^{2^r t} \equiv -1 \bmod p_2$ is, by the independence, bounded by $2^{r - r_1} 2^{r - r_2} = 2^{2r - r_1 - r_2}$ if $r < r_2$ and 0 otherwise. Thus the probability of having -1 simultaneously for *some* r is at most

$$\sum_{r=0}^{r_2 - 1} 2^{2r - r_1 - r_2} \tag{3.2}$$

Finally, let us take into account the other possible event, (i). The probability that

$$u_0 \equiv 1 \bmod p_1 \qquad \text{and} \qquad u_0 \equiv 1 \bmod p_2.$$

is, by Lemma 3.10, bounded by $2^{-(r_1 + r_2)}$. We need to bound

$$2^{-(r_1 + r_2)} + \sum_{r=0}^{r_2 - 1} 2^{2r - r_1 - r_2}.$$

Replacing on of the two r's in the second exponent by its maximal value gives the upper estimate

$$2^{-(r_1 + r_2)} + \sum_{r=0}^{r_2 - 1} 2^{r - (r_1 + 1)}.$$

The sum is equal to $2^{r_2 - (r_1 + 1)} - 2^{-(r_1 + 1)}$ and since $r_1 \geq r_2 \geq 1$ we have the total estimate

$$2^{r_2 - (r_1 + 1)} \leq 1/2.$$

$\square$

3.2.1 Some notes on the algorithm

If there are many a's that can be used as witnesses of the compositeness of N, can we say something about a in advance? Is there some set of a's that *guarantees* revealing the compositeness?

There exists a number theoretic conjecture known as the *extended Riemann hypothesis*, ERH. (See [26] for details.)

Theorem 3.11. *Under the ERH, there always exists a witness $a \le 2(\log N)^2$ of the compositeness of N.*

Corollary 3.12. *Under the ERH, the Miller-Rabin test can be converted to a* deterministic *test of primality that runs in time $O((\log N)^5)$.*

Simply try all $a \le 2(\log N)^2$.

3.3 Other tests for primality

There are several other, more or less practical, tests. We list a few by their inventors. Below, n equals $\log_2 N$.

Solovay & Strassen, [51]. Define the *Legendre symbol* $\left(\frac{b}{p}\right) = 1$ if the equation $x^2 \equiv b \bmod p$, p being prime, has a solution, -1 otherwise. The *Jacobi symbol* $\left(\frac{b}{N}\right)$ when $N = \prod p_i^{e_i}$ is defined as the product of the Legendre symbols $\left(\frac{b}{p_i}\right)^{e_i}$ for each prime factor p_i in N. The Jacobi symbol can be efficiently computed even if the factorization of N is unknown.

It can be shown that *if* N is prime then

$$b^{(N-1)/2} \equiv \left(\frac{b}{N}\right) \bmod N \tag{3.3}$$

but if N is composite then the probability that (3.3) holds for a random b is at most $1/2$. This criterion is in fact stronger than the criterion in Fermat's theorem, since there are no analogs of Carmichael numbers with respect to this test.

Adleman, Pomerance & Rumely, [2]. This is a deterministic algorithm that uses Gauss-sums and runs in time $O(n^{\log\log n})$. The algorithm is not practical.

Cohen & Lenstra, [11]. This algorithm is based on the same ideas as the one by Adleman, Pomerance and Rumely, but uses Jacobi sums which makes it much more efficient in practice.

Goldwasser & Killian, [22]. This is a probabilistic algorithm that *always* gives a correct answer. It runs in expected polynomial time. It is provably correct for "most" p. The basic idea is simple. It can be described as: "If a has order $(N-1)/2$ mod N then: $(N-1)/2$ is prime implies N is prime." As an example, 4 has order 5 mod 11 so if 5 is a prime then 11 is prime too! The crux is that it is very unlikely that $(N-1)/2$ be prime. As an example, the statement "If 8 is prime then 17 is prime" is of no great help. This problem was circumvented by Goldwasser & Killian by working with elliptic curves instead of the integers mod N.

Adleman & Huang, [1]. This is a "fix" of Goldwasser & Killian that makes it work for all N. It uses so called Jacobians of elliptic curves. The idea is slightly counter-intuitive. It reduces the problem of deciding primality for a certain number N to the problem of deciding primality for a number N' that is *larger* than N ($N' \sim N^2$). It would seem that nothing is gained by this, but what happens is that N' is somewhat random and it is possible to run Goldwasser & Killian on this number instead.

Chapter 4

Factoring integers

So, suppose that we have fed N to some algorithm for primality testing and that we got the answer "N is composite". This gives no information on the factors of N and obtaining these turns out to be a more difficult problem.

The motivation for this problem is the same as for primality. Note here that for application to the RSA cryptosystem it is important that there are no really efficient algorithms for factoring while for the primality problem the existence of efficient algorithm was essential.

4.1 The "naive" algorithm

If N is composite, it must have at least one prime factor p with $p \leq \sqrt{N}$. We try dividing N by all integers $p \leq \sqrt{N}$. The running time is similar to the size of the prime factor p found which can be as large as $\Omega(\sqrt{N}) = \Omega(2^{n/2})$. This is clearly very inefficient if n is large.

4.2 Pollard's ρ-method

We start by giving the algorithm first and then discussing it. The algorithm finds one non-trivial factor $d|N$. If the factors are not found to be prime by the methods of the previous section the algorithm is used again until the found factors are prime numbers.

Pollard(N) =
  let $x_0 \in_U \mathbb{Z}_N$
  define the sequence $\{x_i\}_{i\geq 1}$ by $x_{i+1} \equiv x_i^2 + 1 \bmod N$
  for $i := 1, 2, \ldots$ **do**
    $d := \gcd(x_{2i} - x_i, N)$
    if $d > 1$ **then**
      d is a non-trivial factor in N, **stop**

The big questions are: How can we be sure that we find a non-trivial factor and if so, how long do we have to wait?

We cannot (and nobody else can for that matter) give a mathematically exact answer to this question and hence we resort to a heuristic argument.

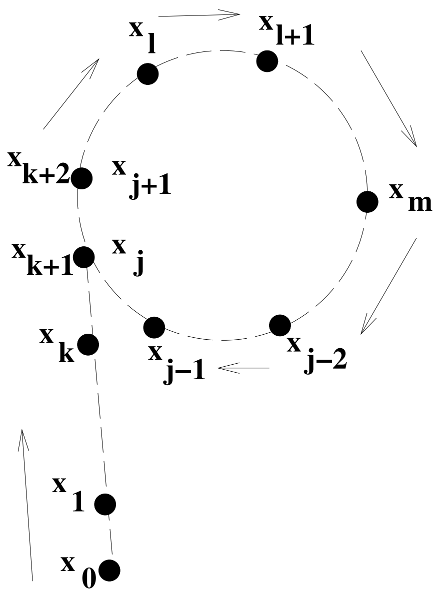

Figure 4.1: The sequence $x_i \bmod p$.

Assume that $p|N$. If the sequence $\{x_i\}_{i\geq 1}$ behaves "randomly" so that x_{2i}, x_i can be considered random integers mod N, then one would expect

$$\Pr[p \text{ divides } x_{2i} - x_i] = \frac{1}{p}$$

and thus it would take about p steps to find the factor p. This is no better than the naive trial division algorithm and in fact this intuition is incorrect.

Claim 4.1. *If the mapping $x_{i+1} \equiv x_i^2 + 1 \bmod N$ was replaced by a mapping $x_{i+1} = f(x_i)$ for a random function f mod p the factor p is found after $O(\sqrt{p}) \in O(N^{1/4})$ steps.*

Although it has not been proved, experience shows that the updating rule $x_{i+1} \equiv x_i^2 + 1$ also behaves this way.

Proof. Let us study the sequence $\{x_i\}_{i\geq 1}$ mod p. We do this graphically, in Figure 4.1[1]. After the j:th step, the sequence repeats itself! We can look at it as having two runners, x_{2i} and x_i running in a circle, one with twice the speed of the other. Eventually x_{2i} catch up with x_i. It is when this passing occurs that the factor p falls out; since $x_{2i} \equiv x_i \bmod p$, $p|x_{2i} - x_i$.

What remains is to estimate the number of steps until this occurs, the "circumference" of the circle. This follows from the lemma below. □

Lemma 4.2. *If $f : \mathbb{Z}_p \mapsto \mathbb{Z}_p$ is a random function and we define the sequence $\{x_i\}_{i\geq 0}$ by $x_{i+1} = f(x_i)$ with an arbitrary x_0, we have:*

$$E_f[\text{ smallest } j \text{ s.t. } x_j \equiv x_i \bmod p,\ i < j] \in O(\sqrt{p}),$$

[1] The name of this algorithm should be obvious from this figure. The Greek letter ρ is pronounced "rho".

where E_f is the expectation operator with respect to a random f.

Proof. Exercise. □

As the algorithm is written it seems like we need to remember all the iterates between x_i and x_{2i}. It is however, much more efficient to only remember two values and recompute the value of x_i whenever it is needed rather than to remember it.

Could we have chosen f in some other way, say by computing $x_{i+1} = \lfloor N|\sin(2\pi x_i/N)|\rfloor$? The answer is no! Our argument was based on studying x_i mod p and thus we need the property that x_{i+1} mod p is determined by x_i mod p (and not all of x_i). This is not true when $x_{i+1} = \lfloor N|\sin(2\pi x_i/N)|\rfloor$.

Why the choice of the specific function $x \mapsto x^2 + 1 \bmod N$? The reason for this is basically heuristic: It works well in practice, and it is also easy to compute. An essential condition is also that the function is not $1-1$ since such functions would lead to running times of order p.

Let us end with an example of an execution of the ρ-method.

Example 4.3. *Let us factor $N = 1387 = 19 \cdot 73$. Start by $x_0 = 0$. Let $y_i \equiv x_{2i} - x_i \bmod 1387$.*

i:	1	2	3	4	5	6	7	8	9	10
x_i	1	2	5	26	677	620	202	582	297	829
x_{2i}	2	26	620	582	829	$\cdots$				
y_i	1	24	615	556	152	$\cdots$				
$\gcd(y_i, 1387)$	1	1	1	1	19	$\cdots$				

After step 5, the factor 19 falls out. Observe that (of course, mostly by coincidence) $\sqrt{19} \approx 4.36$.

4.3 A method used by Fermat

The basis of this, and many other methods, is to find a solution to the equation $x^2 \equiv y^2 \bmod N$ where $x \not\equiv \pm y \bmod N$. The reason this is useful is that if the equations is satisfied then

$$0 \equiv x^2 - y^2 \equiv (x+y)(x-y) \quad mod\ N,$$

and since neither factor is 0 mod N, $gcd(x+y, N)$ is nontrivial and we find a factor in N.

Fermat's approach was to find q's such that q^2 was small mod N. The idea is that it is more likely that a small number is a square. Let us estimate this probability heuristically. Since the distance between s^2 and $(s+1)^2$ is about $2s$ the probability that a random number of size roughly x is a square is about $\frac{1}{2\sqrt{x}}$.

One approach for getting numbers q with q^2 small mod N is choosing an integer of the form $q = \lceil\sqrt{aN}\rceil + b$ where a and b are small integers. Then $q = \sqrt{aN} + b + \delta$ for some real number δ where $0 < \delta < 1$ and hence

$$q^2 = aN + (b+\delta)\sqrt{aN} + (b+\delta)^2 \equiv (b+\delta)\sqrt{aN} + (b+\delta)^2 \mod N$$

This is, for constant size a and b of size $O(\sqrt{N})$. Heuristically the probability that it is a square is $\Omega(N^{-1/4})$ and thus we get a factoring algorithm that heuristically runs in time $O(N^{1/4})$.

Example 4.4. *Suppose we want factor* 561*. We try integers slightly larger than* $\sqrt{561}$ *and get*

$$\begin{aligned} 24^2 &= 576 \equiv 15 \\ 25^2 &= 625 \equiv 64. \end{aligned}$$

Numbers slightly larger than $\sqrt{2 \cdot 561}$ *give no squares while trying* $3 \cdot 561$ *results in* $42^2 = 1764 \equiv 81$*. The relation* $25^2 \equiv 8^2$ *tells us to compute* $gcd(561, 25 - 8) = 17$ *or* $gcd(561, 25 + 8) = 33$ *to get nontrivial factors. In the same way* $gcd(561, 42 - 9) = 33$ *and* $gcd(561, 42 + 9) = 51$ *gives what we desire.*

There is no known general method to, without knowing the factorization of N, generate $Q > \sqrt{N}$'s such that $Q^2 \bmod N$ is significantly smaller than $\sqrt{N}$.

4.4 Combining equations

Consider the following. Let $N = 4633$ be a number to be factored. Then

$$67^2 \equiv -144 \mod N$$

and

$$68^2 \equiv -9 \mod N.$$

Neither of these equations can be used directly to obtain a factorization of N. However, they can be combined to yield

$$(67 \cdot 68)^2 \equiv -144 \cdot -9 \equiv= (12 \cdot 3)^2 \mod N.$$

Since $67 \cdot 68 \equiv (-77) \bmod N$ we obtain the factorization $4633 = 41 * 113$ since $gcd(-77 + 36, 4633) = 41$ and $gcd(-77 - 36, 4633) = 113$.

We want to expand this to a more general approach. Take a number B and consider all primes smaller than B together with the special "prime" -1. These primes make up our *factor base* . We generate a number Q such that $Q^2 \bmod N$ has a small absolute value and hope that it factors in the factor base. If it does we remember it and pick a new Q and try again. This way we generate equations of the form

$$Q_i^2 \equiv \prod_{p<B} p^{a_{i,p}} \bmod N$$

for some $i = 1, 2 \ldots s$. We want to combine these by choosing a suitable subset S and multiplying the equations in this subset. This yields an equation of the form

$$\prod_{i \in S} Q_i^2 \equiv \prod_{i \in S, p<B} p^{a_{i,p}} \equiv \prod_{p<B} p^{\sum_{i \in S} a_{i,p}} \bmod N.$$

The left hand side is always a square while the right hand side is a square if and only if $\sum_{i \in S} a_{i,p}$ is even for any p. Thus all that remains is to find such a subset S. However this is just linear algebra mod 2. In particular, we start with the vectors

$$\begin{array}{cccc}
(a_{1,-1}, & a_{1,2}, & a_{1,3}, & \dots a_{1,B}) \\
(a_{2,-1}, & a_{2,2}, & a_{2,3}, & \dots a_{2,B}) \\
(a_{3,-1}, & a_{3,2}, & a_{3,3}, & \dots a_{3,B}) \\
\vdots & \vdots & \vdots & \vdots \\
(a_{s,-1}, & a_{2,2}, & a_{s,3}, & \dots a_{s,B})
\end{array}$$

where each coefficient is considered a number mod 2 and we want to find a linear dependence. If there are more vectors than dimensions there is always such a dependence and it can be found efficiently by Gaussian elimination. Combining this set of equations gives a relation $x^2 \equiv y^2 \bmod N$ as described above. There is no guarantee that it is nontrivial (*i. e.* $x \not\equiv \pm y$) and there is not even a proof that it is nontrivial with reasonable probability. However, heuristically it should be nontrivial with probability 1/2 and this turns out to be true in practice.

There are $\pi(B) \approx \frac{B}{\ln B}$ primes smaller than B and thus we need at least that many equations. This leads to the following algorithm which is parameterized by B.

1. Find all primes $\leq B$.
2. Find numbers Q of the form $\lceil \sqrt{aN} \rceil + b$ where a and b are small and see if $Q^2 \bmod N$ factor in primes $\leq B$. Repeat until we have $\pi(B)+2$ successes.
3. Find a subset S of equations by Gaussian elimination such that when the corresponding equations are multiplied we get a relation $x^2 \equiv y^2 \bmod N$. If this is trivial (i.e. $x \equiv \pm y \bmod N$) then find another number Q such that $Q^2 \bmod N$ factors in the factor base and repeat until we get a nontrivial relation (and hence a successful factorization).

Let us try to analyze this algorithm. One key to performance is the choice of B. Let $S(A, B)$ be the probability that a random number of size around A factors in prime factors bounded by B. Then heuristically the second step runs in time about

$$B^2 S(\sqrt{N}, B)^{-1}.$$

This follows since we need to find approximately B numbers that factor and for each success we expect to examine $S(\sqrt{N}, B)^{-1}$ numbers and to investigate whether one number factors can be done in time about B. The third step runs in time $O(B^3)$. To balance the factors above B should be chosen around $2^{c\sqrt{\ln N \ln\ln N}}$ for a suitable constant c which gives a running time of $e^{\sqrt{\ln N \ln\ln N}}$. In practice one has to choose B by experience. In the recent success in factoring a 129 digit number B was chosen to be around 10^7.

Example 4.5. *Suppose we want to factor 85907 and we choose $B = 10$ as the maximal size of primes to be considered. We take integers Q close to $\sqrt{a \cdot 85907}$ for small integers a and try to factor Q^2 mod 85907 using the primes -1,2,3,5 and 7. The result is*

$$\begin{aligned}
501^2 &\equiv -6720 \equiv -1 \cdot 2^6 \cdot 3 \cdot 5 \cdot 7 \\
507^2 &\equiv -672 \equiv -1 \cdot 2^5 \cdot 3 \cdot 7 \\
508^2 &\equiv 343 \equiv 7^3 \\
829^2 &\equiv -15 \equiv -1 \cdot 3 \cdot 5.
\end{aligned}$$

We now find linear relations among the rows of the matrix of exponents mod 2, i.e., of

$$\begin{pmatrix} 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \end{pmatrix}.$$

In general we could not hope for such a relation since we have more variables than equations, but this time we are lucky and find that the sum of rows 1, 3 and 4 is 0. Of course, in general, we would find such a relation by Gaussian elimination. We multiply the three equations and obtain

$$(501 \cdot 508 \cdot 829)^2 \equiv (2^3 \cdot 3 \cdot 5 \cdot 7^2)^2$$

or $85447^2 \equiv 5880^2$*. We can then find the factors as* $gcd(85447 + 5880, 85907) = 271$ *and the cofactor is* 317.

4.4.1 Implementation tricks

To make the given algorithm run fast a number of implementation tricks should be used and let us mention one. Suppose we fix a. Whether $(\lceil \sqrt{aN} \rceil + b)^2$ mod N is divisible by a specific prime p depends only on b mod p. This suggests the following algorithm

1. Store a floating point approximation of $\log Q_b$ where $Q_b = (\lceil \sqrt{aN} \rceil + b)^2$ mod N in cell b of an array.
2. For each p find possible values of b mod p such that Q_b is divisible by p. There are either no such b or two possible values which we call b_1 and b_2. In the latter case subtract a floating point approximation of $\log p$ from cells with addresses $b_1 + ip$ and $b_2 + ip$ for all possible values of i.
3. For cells b that contain a value close to 0 try to factor Q_b into small primes.

If p is small we also need to mark the numbers divisible by p^2 or higher powers of p in a similar manner. The key in the above procedure is that the expensive step 3 is only carried out when we are likely to succeed. The given implementation is very similar to a sieve and the algorithm which was discovered by Pomerance is called the *quadratic sieve*, see [40], [20].

4.5 Other methods

There are a number of other methods for factoring integers and let us mention their names and a couple of sentences on each method.

1. **Elliptic curves, [32], [37].** Generate a random elliptic curve and hope that the number of points on the curve is a number that only has small prime factors. This algorithm runs in expected time

$$e^{d\sqrt{\ln N \ln \ln N}}$$

   for some constant d. This algorithm has the property that it finds small prime factors faster than large prime factors.

2. **Continued fractions, [38].** This algorithm is similar to the quadratic sieve. It also uses a factor base and the key difference is that it generates the numbers $Q^2 \bmod N$ to be factored using continued fractions as described in Section 4.5.1 below. This algorithm also runs in time
$$e^{d\sqrt{\ln N \ln\ln N}}$$
for some constant d.

3. **Number field sieve, [33].** This algorithm was originally designed to factor numbers N of the form $a^b + c$ where a and c are small. Suppose we want to factor $7^{200} + 11$. Let $\alpha = 7^{40}$. Then $\alpha^5 = -11 \bmod N$ and thus can be considered as an algebraic number. Also this algorithm has a factor base. It contains the small standard primes but also numbers of the form
$$a_0 + a_1\alpha + a_2\alpha^2 + a_3\alpha^3 + a_4\alpha^4$$
where this is a prime with small norm when considered as an algebraic integer. The algorithm then generates numbers of the form $a\alpha + b$ and tries to factor them both as a standard integer and as an algebraic number. As usual it collects successful factorizations and then finds a dependence mod 2. Later developments of this algorithm has made it applicable to general numbers and it is today the most efficient general methods for large numbers. In 1999 a 151 digit number was factored using this algorithm and it is currently considered the champion for factoring large numbers. The expected running time of the algorithm is, by a heuristic argument showed to be
$$O(e^{1.9223(\ln N)^{1/3}(\ln\ln N)^{2/3}}$$
For a more current comparison of the two factoring algorithms we refer to [41]. Let us give an example below.

Example 4.6. *We factor $N = 20437$ using the number field sieve. When using NFS we first find a polynomial $p(x)$ together with a number $m \in \mathbb{Z}$ such that $p(m) = 0 \bmod N$. Let α be a complex number such that $p(\alpha) = 0$ over the complex numbers. Suppose p is of degree d and let us study numbers of the form $\sum_{i=0}^{d-1} a_i\alpha^i$ where $\alpha_i \in \mathbb{Z}$. Since we can use the equation $p(\alpha) = 0$ to replace α^i for $i \geq d$ by lower powers of α we can multiply numbers of this form and get a new number of the same form. They form what is called a ring of algebraic integers. The point is now that replacing α by m we get a homomorphism from these algebraic integers into $\mathbb{Z}_N$. Note that for this to work it is crucial that $p(m) \equiv 0 \bmod N$.*

We can construct p using the base-m method, that is, writing N to the base m. Let $m = \llcorner N^{1/3} \lrcorner - 1 = 26$ and write $20437 = 26^3 + 4 \cdot 26^2 + 6 \cdot 26 + 1$ giving $p(\alpha) = \alpha^3 + 6\alpha^2 + 4\alpha + 1$ which has the desired property.

We want to factor a lot of algebraic numbers. Just as numbers in $\mathbb{Z}$ are products of primes, the algebraic numbers are products of irreducible[2] algebraic

[2] This is unfortunately only partly true. There is a small problem that there might be units (numbers invertible in the ring different from ± 1) but the large problem is that factorization on this level might be non-unique for some (or even most) choices of p. We sweep this problem under the rug. We can get unique factorization by factoring into ideals rather than numbers, where a number corresponds to a principal ideals. The reader who is not comfortable with ideals can simply ignore this footnote.

numbers. The ideas is, for small a and b, to factor $a + bm$ as a normal integer and $a + b\alpha$ as an algebraic integer. We get an interesting identity mod N and we later combine such identities to get a solution to $x^2 \equiv y^2 \bmod N$.

When factoring ordinary integers we use a standard factor base (in our case of primes upto 13 together with -1, i.e. $\mathcal{P}_{1,2,\ldots,7} = \{-1, 2, 3, 5, 7, 11, 13\}$) and when factoring the algebraic integers we choose a small set of algebraic numbers (which we hope are algebraic primes although, in all honesty, we have not checked this). We have settled for $\mathcal{A}_{1,2,\ldots,6} = \{-\alpha, -1-\alpha, 2-\alpha+\alpha^2, -2+3\alpha+\alpha^2, 3+\alpha, -5-3\alpha-\alpha^2\}$.

We now factor numbers of the form $a + b\alpha$ into algebraic primes and at the same time we factor $a + bm$ into ordinary primes.

An example would be

$$4 + 2\alpha = (-1-\alpha)(-5-3\alpha-\alpha^2) = \mathcal{A}_2\mathcal{A}_6, \tag{4.1}$$

where we can check the factorization by simply multiplying the factors obtaining

$$\begin{aligned}(-1-\alpha)(-5-3\alpha-\alpha^2) &= \\ 5+3\alpha+\alpha^2+5\alpha+3\alpha^2+\alpha^3 &= \\ 5+3\alpha+\alpha^2+5\alpha+3\alpha^2-(1+6\alpha+4\alpha^2) &= 4+2\alpha\end{aligned}$$

Inserting $m = 26$ in (4.1) gives $(\mathcal{A}_2\mathcal{A}_6)(m) = 4 + 2 \cdot m = 4 + 2 \cdot 26 = 56 = 2^3 \cdot 7$. Note that both factors 2 and 7 is in the prime factor base $\mathcal{P}$. If we were to find a product with at least one factor outside $\mathcal{P}$ it would be rejected. E.g. $\mathcal{A}_5(m) = 3 + m = 3 + 26 = 29$ which is prime but outside $\mathcal{P}$, and thus $\mathcal{A}_5$ would be rejected.

Each accepted product constitutes a row in an exponent matrix. Each element in the factor base corresponds to one column. In each entry of the row, the exponent of the corresponding factor is entered. Take $\mathcal{A}_2\mathcal{A}_6$ and (4.1) as an example: Writing our prime factor base first we get 0300100 since there are three 2:s and one 7. Writing our algebraic factor base second we get 010001 since $\mathcal{A}_2 = (-1-\alpha)$ and $\mathcal{A}_6 = (-5-3\alpha-\alpha^2)$ were factors of $4 + 2\alpha$. This gives us a row 0300100 010001 *in the matrix. We try to factor many numbers of the form $a + b\alpha$, (and $a + bm$) and keeping our successful factorization we get a matrix of exponents:*

$$\begin{pmatrix} 1100001\ 100000 \\ 1030000\ 010000 \\ 1010100\ 020020 \\ 0300100\ 010001 \\ 1120000\ 030001 \\ 0100002\ 000201 \\ 0120000\ 100011 \\ 1220000\ 011002 \\ 0320001\ 111002 \\ 0250000\ 021002 \\ 1322010\ 301102 \\ 0230100\ 031022 \\ 1230100\ 101032 \\ 1320100\ 021003 \end{pmatrix}$$

Here we find our previous example in the fourth row. Another example would be $5 - \alpha = (-1-\alpha)^2(3+\alpha)^2 = \mathcal{A}_2^2 \cdot \mathcal{A}_5^2$. This corresponds to row three:

1010100 020020, *since* $5 - m = 5 - 26 = -21 = -1 \cdot 3 \cdot 7$. *That is, 1010100 for* -1, 3, *7 and 020020 for* $(-1-\alpha)^2$ *and* $(3+\alpha)^2$.

From the exponent-matrix we want to find integers x, y *such that* $x^2 \equiv y^2 \bmod N$. *This is done by finding a linear dependency mod 2. When one such is found, the* x *is constructed from the prime factor base and the* y *is created in the algebraic factor base. The procedure of the construction is explained below. First, let us find a dependency.*

We create a boolean matrix, taking the exponent-matrix elements mod 2. We augment the matrix by an identity matrix for later use:

$$\begin{pmatrix}
1100001\ 100000\ 10000000000000 \\
1010000\ 010000\ 01000000000000 \\
1010100\ 000000\ 00100000000000 \\
0100100\ 010001\ 00010000000000 \\
1100000\ 010001\ 00001000000000 \\
0100000\ 000001\ 00000100000000 \\
0100000\ 100011\ 00000010000000 \\
1000000\ 011000\ 00000001000000 \\
0100001\ 111000\ 00000000100000 \\
0010000\ 001000\ 00000000010000 \\
1100010\ 101100\ 00000000001000 \\
0010100\ 011000\ 00000000000100 \\
1010100\ 101010\ 00000000000010 \\
1100100\ 001001\ 00000000000001
\end{pmatrix}$$

Dependencies are found using Gaussian elimination in our matrix. Addition of rows mod 2 reduces to row-wise exclusive or, yielding

$$\begin{pmatrix}
1100001\ 100000\ 10000000000000 \\
0110001\ 110000\ 11000000000000 \\
0010101\ 100001\ 11010000000000 \\
0000100\ 010000\ 01100000000000 \\
0000011\ 001100\ 10000000001000 \\
0000001\ 111001\ 11110001000000 \\
0000000\ 100010\ 01110010000000 \\
0000000\ 001000\ 01111001000000 \\
0000000\ 000000\ 10000001100000 \\
0000000\ 000000\ 01000001010000 \\
0000000\ 000000\ 01110100000000 \\
0000000\ 000000\ 00100001000100 \\
0000000\ 000000\ 00101011000010 \\
0000000\ 000000\ 00010001000001
\end{pmatrix}$$

We note that the last six rows contains only zeros in the factor base part of the matrix, that is, even exponents of the factors. The augmented part of the matrix gives us the information which rows were added to obtain each dependency. For example, 00010001000001 tells us that the last row is the sum of rows 4, 8 and 14 in the exponent-matrix.

Thus multiplying original rows 4, 8, and 14 gives us both a square integer and a square algebraic number. The square integer will be our x^2. *If we call the algebraic square* B^2, y *will be given by* $y = B(m)$ *evaluated just as in the example with (4.1). We do two examples to highlight some different cases.*

First we evaluate our first row of zeros: 10000001100000. This corresponds to a product of rows 1, 8 and 9:

$$\begin{array}{rl} 1: & 1100001\ 100000 \\ 8: & 1220000\ 011002 \\ 9: & 0320001\ 111002 \\ \text{column sum}: & 2640002\ 222004 \end{array}$$

As expected all sums are even and we have a square integer and a square algebraic number. We divide each exponent in 2640002 by two and get 1320001 giving $x = -1 \cdot 2^3 \cdot 3^2 \cdot 13 = -936$. *Half of 222004 is 111002 which in turn implies* $B = (-\alpha)(-1-\alpha)(2-\alpha+\alpha^2)(-5-3\alpha-\alpha^2)^2 = 36\alpha$. *Replacing* α *by* m *gives* $B(m) = 36 \cdot m = 36 \cdot 26 = 936$. *In this case we have* $x = -y \bmod N$ *which makes the result useless to factor* N.

We now try our third row of zeros: 01110100000000. This tells us that we should take the product of rows 2, 3, 4 and 6:

$$\begin{array}{rl} 2: & 1030000\ 010000 \\ 3: & 1010100\ 020020 \\ 4: & 0300100\ 010001 \\ 6: & 0100002\ 000201 \\ \text{column sum}: & 2440202\ 040222. \end{array}$$

Half of 2440202 is 1220101 giving $x = -1 \cdot 2^2 \cdot 3^2 \cdot 7 \cdot 13 = -3276$. *Half of 040222 is 020111, yielding* $B = (-1-\alpha)^2(-2+3\alpha+\alpha^2)(3+\alpha)(-5-3\alpha-\alpha^2) = 26 + 30\alpha + 16\alpha^2$.

This time insertion gives $B(m) = 26 + 30 \cdot m + 16 \cdot m^2 = 26 + 30 \cdot 26 + 16 \cdot 26^2 = 11622$, *and* x, y *is not on the form* $x \equiv \pm y \bmod N$. *We now calculate* $\gcd(x + y, N) = \gcd(-3276 + 11622, 20437) = \gcd(8346, 20437)$ *which will be a nontrivial factor of 20437. The Euclidean algorithm gives*

$$\begin{aligned} 20437 &\equiv 3745 \bmod 8346 \\ 8346 &\equiv 856 \bmod 3745 \\ 3745 &\equiv 321 \bmod 856 \\ 856 &\equiv 214 \bmod 321 \\ 321 &\equiv 107 \bmod 214 \\ 214 &\equiv 0 \bmod 107, \end{aligned}$$

yielding $\gcd(8246, 20437) = 107$. *We also note that* $\gcd(x-y, N) = \gcd(-14898, 20437) = 191$ *and in fact* $20437 = 107 \cdot 191$. *Of course the factor* 191 *could also have been obtained by a division.*

4.5.1 Continued fractions

Continued fractions is a piece of classical mathematics. It is a basic construct of efficient computation, since viewed carefully it is very similar to the Euclidean algorithm (consider what happens when you apply it to a rational number).

Continued fractions is a method for obtaining good rational approximations to a real number Θ. Suppose $\Theta > 1$, then we write

$$\Theta = a_1 + \frac{1}{\Theta_1} \tag{4.2}$$

where a_1 is the largest integer smaller than Θ and Θ_1 is whatever makes the equation correct. By construction, $\Theta_1 > 1$ and hence we can repeat the process with Θ_1 etc

$$\Theta = a_1 + \frac{1}{a_2 + \frac{1}{\Theta_2}} = a_1 + \frac{1}{a_2 + \frac{1}{a_3 + \frac{1}{\Theta_3}}}.$$

Replacing $\frac{1}{\Theta_i}$ by 0 gives a rational number $\frac{p_i}{q_i}$ which turns out to be a good approximation of Θ. For instance the famous approximation $\frac{22}{7}$ for π can be obtained by using

$$\pi = 3 + \frac{1}{7 + \frac{1}{\pi_2}}$$

and then ignoring $\frac{1}{\pi_2}$. In general it is true that

$$|\Theta - \frac{p_i}{q_i}| \leq \frac{1}{q_i^2}. \tag{4.3}$$

The numbers p_i and q_i can be obtained very efficiently given a floating point approximation of Θ. In our case $\Theta = \sqrt{N}$ and then all Θ_i can be written on the form $\frac{\sqrt{N}+b}{c}$ making the calculations even simpler.

Let us see how to use this to obtain numbers Q such that Q^2 are small. We use the continued fraction expansion of $\sqrt{N}$ and let $\frac{p_i}{q_i}$ be one obtained fraction. Since the derivative of x^2 at $\sqrt{N}$ is $2\sqrt{N}$ we have by (4.3)

$$|\frac{p_i^2}{q_i^2} - N| \approx 2\sqrt{N}|\frac{p_i}{q_i} - \sqrt{N}| \leq \frac{2\sqrt{N}}{q_i^2}.$$

This implies that

$$|p_i^2 - q_i^2 N| \leq 2\sqrt{N}$$

and hence we can use p_i as our numbers with small squares mod N.

Example 4.7. *Let us do the continued fraction expansion of $\theta = \sqrt{45}$. We have using (4.2) that $a_1 = 6$ and*

$$\theta_1 = \frac{1}{\sqrt{45}-6} = \frac{\sqrt{45}+6}{9},$$

where we multiplied both numerator and denominator by $\sqrt{45}+6$ and used the conjugate rule. We get $a_2 = 1$ and

$$\theta_1 = \frac{9}{\sqrt{45}-3} = \frac{\sqrt{45}+3}{4},$$

which gives $a_3 = 2$ and

$$\Theta_3 = \frac{4}{\sqrt{45}-5} = \frac{\sqrt{45}+5}{5}$$

and hence also $a_4 = 2$. Truncating gives the approximation

$$6 + \frac{1}{1 + \frac{1}{2+\frac{1}{2}}} = \frac{47}{7} \approx 6.714$$

which is a reasonable approximation since $\sqrt{45} \approx 6.708$.

Chapter 5

Discrete Logarithms

Let us start by introducing some notation. Z_m is the set of integers mod m and Z_m^* is the set of invertible elements (i.e. for all $a \in Z_m$ such that there exists $b \in Z_m$ such that $ab \equiv 1 \bmod m$.). It is not hard [1] to see that Z_m^* are all the elements such that $gcd(a, m) = 1$. We are primarily interested in the case when m is a prime p and in this case Z_p^* consists of all the integers $1, 2, \ldots, p-1$.

A number g is called a *generator* if the series $g, g^2, g^3, \ldots, g^{p-1} \bmod p$ includes all numbers $1, \ldots, p-1$. It is well known that for each prime p there exists a generator (we do not prove this). Another way to formulate this is to say that the multiplicative group $\mathbb{Z}_p^*$ is cyclic.

Example 5.1. *The multiplicative group with $p = 13$ is generated by $g = 2$ since the powers of 2 are:*

$$2, 4, 8, 16 \equiv 3, 6, 12, 24 \equiv 11, 22 \equiv 9, 18 \equiv 5, 10, 20 \equiv 7, 14 \equiv 1.$$

With $g = 4$, only a subgroup is generated.

$$4, 16 \equiv 3, 12, 48 \equiv 9, 36 \equiv 10, 40 \equiv 1.$$

There are many generators g, but there is no known way to find such a generator in deterministic polynomial time. Even worse, even testing whether a given number is a generator is not known to be in deterministic polynomial time. Things are not as bad as they seem, however, and in polynomial time one can find a number that is very likely to be a generator and we assume that we are given such a number.

The multiplicative group (mod p), $\mathbb{Z}_p^*$, is isomorphic with the additive group (mod $p-1$), $\mathbb{Z}_{p-1}$. The mapping is $g^x \in \mathbb{Z}_p^* \leftrightarrow x \in \mathbb{Z}_{p-1}$. Furthermore, $\mathbb{Z}_{p-1}$ is generated by any number a such that $gcd(a, p-1) = 1$. Given such an $a \in \mathbb{Z}_{p-1}$, g^a is a generator of $\mathbb{Z}_p^*$ iff g is a generator of $\mathbb{Z}_p^*$.

Example 5.2. *The generators of the additive group $\mathbb{Z}_{12}$ are $1, 5, 7$ and 11. We know, from example 5.1, that 2 is a generator of the multiplicative group $\mathbb{Z}_{13}^*$. Therefore, the generators of $\mathbb{Z}_{13}^*$ are $2^1 \equiv 2, 2^5 \equiv 6, 2^7 \equiv 11$ and $2^{11} \equiv 7$. Had we known that 11 was a generator, we could have constructed the other generators of $\mathbb{Z}_{13}^*$ as, $11^1 \equiv 11, 11^5 \equiv 7, 11^7 \equiv 2$ and $11^{11} \equiv 6$.*

[1] Think of the extended Euclidean algorithm.

The problem of finding the discrete logarithm can be formulated as follows: Given the numbers y, g and p, where p is prime and g is a generator of $\mathbb{Z}_p^*$, find x such that $g^x \equiv y \bmod p$. This is called the discrete logarithm of y base g modulo p.

Example 5.3. *The discrete logarithm of* 11 *(base* 2 *modulo* 13*) is* 7*. This can be seen from the calculations in example 5.1.*

It is important to remember that discrete logarithms mod p are numbers mod $p-1$. In particular $2^{19} \equiv 11$ mod 13. This is OK since $19 \equiv 7$ mod 12. We could also replace 19 by 31 or 43 without violating the equation.

5.1 A digression

One motivation to study the problem of computing discrete logarithms is given by the Diffie-Hellman [14] key-exchange algorithm from cryptography. We have two parties A and B who want to communicate over an open link to determine a secret common key K. We want K to be secret also to potential eavesdroppers who have listened to the entire communication. This might seem impossible since such a person would seem to have complete information. This is indeed correct but the curious fact is that it might be computationally infeasible to compute the key K. Diffie and Hellman proposed the following protocol.

First a prime p and a generator g is agreed upon and transmitted on the link. Then A transmits g^a, where a is a random number, to B, and B transmits g^b to A for some random number b. The key K is then formed by A as $K \equiv (g^b)^a \equiv g^{ab}$, and by B as $K \equiv (g^a)^b \equiv g^{ab}$. An eavesdropper having only g^a and g^b is at a disadvantage and the only known method to compute K from this information is to compute the discrete logarithm of at least one of them.

5.2 A naive algorithm

A naive algorithm for computing the discrete logarithm of y (base g, modulo p), is to generate $g, g^2, \ldots$ until we get y. However, this is very inefficient, its worst case complexity being $O(2^n)$ if $p \sim 2^n$. That is, in the worst case we have to generate all the powers $g, g^2, \ldots, g^{p-1}$.

5.3 The baby step/giant step algorithm

A somewhat better algorithm for computing the discrete logarithm is the following.

- Let $a = \lceil \sqrt{p} \rceil$
- Compute $L_1 = 1, g^a, g^{2a}, g^{3a}, \ldots, g^{a^2}$
- Compute $L_2 = y, yg, yg^2, yg^3, \ldots, yg^{a-1}$

- Look for a number z which appears in both L_1 and L_2. We then have

$$\begin{aligned} z \equiv yg^k &\equiv g^{la} \bmod p \\ y &\equiv g^{la-k} \equiv g^x \bmod p \\ x &\equiv la - k \bmod (p-1). \end{aligned}$$

The method is called the baby step/giant step algorithm because of the methods for creating the two lists. However, we need to address two issues.

1. Is there always such a number z in both lists?

2. How do we find z efficiently?

The answer to the first question is yes, since we can write $x = x_1 a + x_2$ where $0 \leq x_1, x_2 < a$. We then have $y = g^x = g^{x_1 a + x_2}$. Multiplying both sides of the equation by g^{a-x_2}, we get $yg^{a-x_2} = g^{(x_1+1)a}$. The left hand side of this equation is found in L_2, and the right hand side in L_1.

The answer to the second question is also quite simple. Sort the two lists in time $O(a \log a)$, then proceed as follows: compare the heads of the two lists. If they are equal, we have found z and we stop, otherwise we throw away the smallest of the two front elements and continue. An alternative way is to store L_1 in a hash table, then go through L_2. With a good hash function, this can be done in time $O(a)$ (we return to this question in more detail in section 20.2).

Computing all the numbers in the two lists can be done in $O(a)$ multiplications mod p and since each multiplication can be done in time $O(\log p)^2$ we get the total running time $O((\log p)^2 \sqrt{p})$ which is a substantial improvement over the naive algorithm that requires p multiplications.

Example 5.4. *To solve the equation*

$$2^x \equiv 11 \bmod 13$$

using the algorithm above, we set $a = \lceil \sqrt{13} \rceil = 4$ *and compute*

$L_1 = 1, 2^4 \equiv 3, 2^8 \equiv 9, 2^{12} \equiv 1, 2^{16} \equiv 3$

$L_2 = 11, 11 \cdot 2 \equiv 9, 11 \cdot 2^2 \equiv 5, 11 \cdot 2^3 \equiv 10$

In the two lists we have $11 \cdot 2 \equiv 2^8 \equiv 9$*, which gives us* $11 \equiv 2^7$*. Thus, the logarithm of* 11 *(base* 2 *modulo* 13*) is* 7*.*

5.4 Pollard's ρ-algorithm for discrete logarithms

Pollard also suggested an algorithm for discrete logarithms which uses only a constant number of elements of memory.

We again want to find x such that $g^x \equiv y$ modulo p. The method works as follows:

1. Divide the elements of G into three sets S_1, S_2 and S_3 of roughly equal size. For instance S_i can be the elements x such that $x \equiv i$ modulo 3.

2. Define a sequence of group elements $\{x_i\}$ by:

$$\begin{aligned} x_0 &= 1 \\ x_i &= \begin{cases} yx_{i-1} & \text{if } x_{i-1} \in S_1 \\ x_{i-1}^2 & \text{if } x_{i-1} \in S_2 \\ gx_{i-1} & \text{if } x_{i-1} \in S_3 \end{cases} \end{aligned}$$

   Every element x_i has the form $x_i = g^{a_i} y^{b_i}$, where $\{a_i\}$ and $\{b_i\}$ are explicitly defined by:

$$\begin{aligned} a_0 &= 0 \\ a_i &\equiv \begin{cases} a_{i-1} & \text{if } x_{i-1} \in S_1 \\ 2a_{i-1} & \text{if } x_{i-1} \in S_2 \\ a_{i-1}+1 & \text{if } x_{i-1} \in S_3 \end{cases} \bmod p-1 \\ b_0 &= 0 \\ b_i &\equiv \begin{cases} b_{i-1}+1 & \text{if } x_{i-1} \in S_1 \\ 2b_{i-1} & \text{if } x_{i-1} \in S_2 \\ b_{i-1} & \text{if } x_{i-1} \in S_3 \end{cases} \bmod p-1 \end{aligned}$$

3. Compute the six tuple $(x_i, a_i, b_i, x_{2i}, a_{2i}, b_{2i})$ for $i = 1, 2, \ldots$ until $x_i = x_{2i}$. When $x_i = x_{2i}$ we have that $g^{a_i} y^{b_i} = g^{a_{2i}} y^{b_{2i}}$, or equivalently:

$$\begin{aligned} g^r &= y^s \\ r &\equiv a_i - a_{2i} \bmod p-1 \\ s &\equiv b_{2i} - b_i \bmod p-1 \end{aligned}$$

   This gives us that $\log_g y^s \equiv s \log_g y \equiv r \bmod p-1$. It follows that we have to check at most $d = \gcd(s, p-1)$ values to find the correct value of $\log_g h$. It can be argued heuristically that d is likely to be small, but we omit the details.

If we assume that x_i is a random sequence in G, the running time of the algorithm is by a similar reasoning to that for Pollard's ρ-algorithm for factorization heuristically seen to be $O(\sqrt{p})$. The space requirements is obviously a constant.

5.5 The algorithm by Pohlig and Hellman

We now describe an algorithm due to Pohlig and Hellman Suppose $p-1$ can be factored in "small" primes, *i. e.*

$$p - 1 = \prod_{i=1}^{s} q_i^{\varepsilon_i}.$$

We assume for notational simplicity that $q_1 < \ldots < q_s$ and that $\varepsilon_i = 1$, for $1 \leq i \leq s$. We can then find all q_i efficiently by Pollard's ϱ-algorithm or by using the elliptical curve factoring algorithm which also finds small factors fast. We now compute x "piece by piece". In particular, we want to find an x_i such that $x \equiv x_i \bmod q_i$ and then use the efficient Chinese remainder theorem. Let us see how this is done

For q_1 we first compute

$$(g^x)^{\prod_{i=2}^s q_i} \equiv y^{\prod_{i=2}^s q_i} \stackrel{\text{def}}{\equiv} y_1$$

and

$$(g)^{\prod_{i=2}^s q_i} \stackrel{\text{def}}{\equiv} g_1.$$

The number g_1 generates a subgroup of order q_1 of $\mathbb{Z}_p^*$. This follows since ${g_1}^{q_1} \equiv (g^{\prod_{i=2}^s q_i})^{q_1} \equiv g^{\prod_{i=1}^s q_i} \equiv g^{p-1} \equiv 1$ and since g^{p-1} is the smallest power of g that equals 1 no smaller power of g_1 can equal 1.

Now, writing $x = a_1 q_1 + r_1$ we have

$$\begin{aligned}
(g^x)^{\prod_{i=2}^s q_i} &\equiv (g^{a_1 q_1 + r_1})^{\prod_{i=2}^s q_i} \equiv \\
g^{a_1 \prod_{i=1}^s q_i + r_1 \prod_{i=2}^s q_i} &\equiv g^{a_1(p-1) + r_1 \prod_{i=2}^s q_i} \equiv \\
g^{r_1 \prod_{i=2}^s q_i} &\equiv (g^{\prod_{i=2}^s q_i})^{r_1} \equiv g_1^{r_1} \bmod p.
\end{aligned}$$

Thus all we have to do in order to compute $x \bmod q_1$ is to solve ${g_1}^{r_1} \equiv y_1 \bmod p$. We can do this using $\sim \sqrt{q_1}$ operations on numbers mod p by using the algorithm in Section 5.3 with $a = \lceil \sqrt{q_1} \rceil$.

In the same way, we can determine $x \bmod q_i$ for all q_i in time $\sum_{i=1}^s \sqrt{q_i} \sim \sqrt{q_s}$. The efficient form of the Chinese remainder theorem, then gives us x mod $\prod_{i=1}^s q_i = (p-1)$ and we are done. Since the running time is dominated by the calls to the baby step/giant step part we get the following theorem.

Theorem 5.5. *Given the factorization of $p-1$, discrete logarithms (mod p) can be computed in time $O((\log p)^2 \sqrt{q_s})$, where q_s is the largest prime factor in $p-1$.*

Given p we can apply Pollard's ρ-method to obtain the factorization of $p-1$ in expected time $O((\log p)^2 \sqrt{q_{s-1}})$ where q_{s-1} is the second largest prime factor. Thus finding the factorization of $p-1$ is not the bottleneck.

Example 5.6. *Suppose we want to compute the discrete logarithm of $y = 17$ when $p = 75539$ and $g = 2$. In other words we want to solve*

$$2^x \equiv 17 \bmod 75539.$$

We first note that $p-1 = 75538 = 2 \cdot 179 \cdot 211$ and thus we have $q_1 = 2$, $q_2 = 179$ and $q_3 = 211$. Calculations with respect to q_1 are rather trivial. We have

$$g_1 \equiv g^{75538/2} \equiv g^{37769} \equiv 75538 \equiv -1$$

(as expected since -1 is the only possible generator of a subgroup of size 2). We get

$$y_1 \equiv y^{37769} \equiv 1$$

and hence $x \equiv 0$ mod 2. With q_2 the situation gets a little bit more interesting. We get

$$g_2 \equiv g^{75538/179} \equiv g^{422} \equiv 71077$$

and

$$y_2 \equiv y^{422} \equiv 74852.$$

We compute the two lists as needed in the baby step/giant step algorithm with $a = 14$, $g = g_2$ *and* $y = y_2$. *We get*

$$L_1 = \{1, 53229, 9629, 9926, 31288, 20619, 22620, 23859, 29043, 24212, 9669, 23994, 38753, 39964, 65516\}$$

and

$$L_2 = \{74852, 43834, 58702, 40928, 32566, 27544, 625, 6193, 14108, 49630, 31288, 64555, 61336, 72104\}.$$

Noticing that the 11th number in L_2 *is equal to the fifth number in* L_1 *we have* $y_2 g_2^{10} \equiv g_2^{14\cdot 4} \bmod 75539$ *and thus we can conclude that* $x \equiv 14 \cdot 4 - 10 \equiv 46 \bmod 179$. *Continuing in the same way we get*

$$g_3 \equiv g^{75538/211} \equiv g^{358} \equiv 25883$$

and

$$y_3 \equiv y^{358} \equiv 9430$$

and a similar calculation to the above shows that $x \equiv 12 \bmod 211$. *Finally we combine the 3 modular equations to give* $x \equiv 49808$ *mod* 75538.

5.6 An even faster randomized algorithm

The expected running time of this algorithm is better than for any of the previous algorithms. It is very much similar to the factoring algorithms that use factor bases. The idea is to take a random r_i, compute $g^{r_i} \bmod p$ and hope that $g^{r_i} \equiv \prod_{q<B} q^{a_{i,q}} \bmod p$, where all q are prime and B is an upper limit on the size of the primes. This yields the equation

$$r_i \equiv \sum_{q<B} a_{i,q} \log q, \quad \bmod\ (p-1)$$

where all logarithms are base g, modulo p. Collecting many such relations and solving the resulting system of equations gives us $\log q$ for all $q < B$. Now we pick a new random number r, compute yg^r and hope that it can be factored so that

$$yg^r \equiv \prod_{q<B} q^{b_q} \bmod p.$$

We now have the solution as

$$x = \log y \equiv \sum_{q<B} b_q \log q - r \quad \bmod\ (p-1).$$

The time to generate all the equation needed is

$$Pr[\text{numbers} \sim p \text{ only have factors} \leq B]^{-1} B^2.$$

To solve an equation system with $\sim B$ unknown can be done in time $\sim B^3$. The algorithm works best with $B \sim 2^{c\sqrt{\log p \log\log p}}$, which gives an overall cost of $2^{c'\sqrt{\log p \log\log p}}$. We omit the details.

Example 5.7. *Let us do the calculation as in the previous example, i.e.* $p = 75539$*,* $g = 2$ *and* $y = 17$*. Let us set* $B = 10$ *and thus consider the primes* $2, 3, 5, 7$*. Note that for discrete logarithms* -1 *does not cause any problems since we know that* $g^{(p-1)/2} \equiv -1 \bmod p$ *is always true. We generate some random numbers* r *and obtain*

$$\begin{aligned} g^{9607} &\equiv 1792 = 2^8 \cdot 7 \\ g^{19436} &\equiv 19845 = 3^4 \cdot 5 \cdot 7^2 \\ g^{15265} &\equiv 10584 = 2^3 \cdot 3^3 \cdot 7^2 \\ g^{13835} &\equiv 2250 = 2 \cdot 3^2 \cdot 5^3 \end{aligned}$$

with all equations mod 75538. We get the system

$$\begin{cases} 8\log 2 + & & & \log 7 & \equiv 9607 \\ & 4\log 3 + & \log 5 + & 2\log 7 & \equiv 19436 \\ 3\log 2 + & 3\log 3 + & & 2\log 7 & \equiv 15265 \\ \log 2 + & 2\log 3 + & 3\log 5 & & \equiv 13835 \end{cases}$$

where the system is mod $p - 1$ *i.e. mod 75538. In general we have to solve this by Gaussian elimination but here since* $g = 2$ *we know that* $\log 2 \equiv 1$ *and it turns out that we can solve the system by simply substituting obtained values one by one. Using the first equation and* $\log 2 \equiv 1$ *we get* $\log 7 \equiv 9599$*. Substituting these two values into the third equation yields*

$$3\log 3 \equiv 15265 - 2 \cdot 9599 - 3 \equiv 71602$$

and thus $\log 3 \equiv 74226$*. Plugging this into the last equation gives* $\log 5 \equiv 5486$*. Having computed the logarithms of the small primes we now generate numbers of the form* $yg^r = 17 \cdot 2^r$ *and hope that it factors into our set of primes. We obtain*

$$17 \cdot 2^{31397} \equiv 3024 = 4^2 \cdot 3^3 \cdot 7$$

and we get

$$\log 17 \equiv 4 + 3 \cdot 74226 + \cdot 7599 - 31397 \equiv 49808$$

and we are done.

Note that it is much more expensive to find the logarithms of all small primes than to finally find the discrete logarithm of the number we care about. In particular, computing two discrete logarithms by this method can be done essentially the same cost as computing the discrete logarithm of one number.

Chapter 6

Quantum computation

Classical computers, and classical models of computation, operate using the laws of classical physics. These models can all be simulated by a probabilistic Turing machine in only polynomial cost. In an attempt to find a more powerful model, which could still be physically realizable, it has been suggested to construct a device that follows the laws of quantum mechanics.

The most important evidence that *quantum computers* are indeed more powerful than their classical counterparts is that, if physically realizable, they can do factorization and compute discrete logarithms efficiently [49].

We start this section by giving some quantum mechanical background and describing the quantum circuit model; we then show how to efficiently perform classical computations with such devices. Then, we describe, and partly analyze, the algorithm for integer factorization.

6.1 Some quantum mechanics

A quantum system can be in a number of configurations, called states. Denote a state by x. The complex valued *wave function* $\varphi(x)$ for the system has the following properties.

- $\sum_x |\varphi(x)|^2 = 1$.
- If the system is observed, the probability that we see that it is in the state x is $|\varphi(x)|^2$.
- When the system is observed, the wave function changes. This is due to the added knowledge about the system. In particular, if the whole system is observed to be in the state x, then $\varphi(x) = 1$ and $\varphi(y) = 0$, $y \neq x$.

  Any interaction of the system with the outside world is counted as an observation and thus causes at least a partial collapse of the wave function.

The wave function also has a linear structure as illustrated by the following example.

Example 6.1. *An interesting experiment in quantum physics is the double slit experiment. It consists of an electron source behind a screen. The screen contains two slits, and on its far side is an electron detector. Either slit can be covered.*

If we cover the first slit, the detector registers an electron intensity, $f_1(x)$, for every point x. If the second slit is covered, we similarly get an intensity, $f_2(x)$. However, if both slits are open, the intensity is not, as might first be expected, proportional to $f_1(x) + f_2(x)$.

What happens is that we have two complex valued wave functions, ϕ_1 and ϕ_2, such that $f_1(x) = |\phi_1(x)|^2$ and $f_2(x) = |\phi_2(x)|^2$. The intensity with both slits open is proportional to $|\phi_1(x) + \phi_2(x)|^2$.

In fact, the observed density can not be predicted with knowledge of only $f_1(x)$ and $f_2(x)$, since this does not tell the phase difference of $\phi_1(x)$ and $\phi_2(x)$. Thus, the above complex absolute value can be any real number in the range from $(\sqrt{f_1(x)} - \sqrt{f_2(x)})^2$ to $(\sqrt{f_1(x)} + \sqrt{f_2(x)})^2$.

In analogue to the binary digits of classical computations, quantum computations rely on *qubits* (quantum binary digits). We think of qubits as the memory contents of the quantum computing device. Each qubit may be represented by, for example, an electron which can have up or down spin, or by a photon that is in one of two possible planes of polarization. The physical realization, however, is not important for the theoretical consideration we discuss here.

A system that has m qubits can be in 2^m different states. The wave-function is thus given by 2^m complex numbers $\varphi(\alpha)$, for $\alpha \in \{0,1\}^m$, and we often represent these numbers as a vector of length 2^m. The states themselves are written in *ket* notation as $|\alpha\rangle$, e.g., $|011010\rangle$. As an alternative to vector notation, we sometimes write the wave function as a linear combination of states. For example, $(\sqrt{2}/2)|0\rangle - (\sqrt{2}/2)|1\rangle$ describes the same wave function as does the vector $(\sqrt{2}/2, -\sqrt{2}/2)^T$.

Remember that if we observe the system the probability that we see α is $|\varphi(\alpha)|^2$. Also, as stated above, if we observe the system at any time to be in the state α, the wave function collapses, which means that $\varphi(\alpha) = 1$ while the other coefficients become zero.

Only the part of the system that we observe breaks down; the rest of the system is unaffected. This is natural when we think of the entire world as a quantum system in its own right: just because we observe part of it we do not affect *everything.* More exactly, if we, for instance, observe a single bit, then $\varphi(\alpha)$ either goes to 0 (if α does not agree with the observation) or gets scaled to maintain $\sum_\alpha |\varphi(\alpha)|^2 = 1$. Observing more than one qubit simultaneously can be thought of as observing them one at a time.

6.2 Operating on qubits

We "run" the system of qubits by applying different physical events that form operators on it. An operator is represented by a $2^m \times 2^m$ matrix, $A = (a_{\alpha,\beta})$. Here, α and β are states, and $a_{\alpha,\beta}$ is the wave function $\phi(\alpha)$ after the application of the event, given that the system was in state β. If we represent the wave function of the system by a column vector v of length 2^m, the wave function after the application of an event is described by the matrix multiplication Av.

To make a computation, we repeatedly apply operators to the system in some predetermined order. After some number of operators have been applied, the system is measured; the running time of the computation is the number of operators applied. Notice that the measurement of the system makes the wave function collapse. Therefore, a quantum algorithm cannot apply operators conditionally depending on the state of the system without affecting the computation.

Since the answer given by the measurement is probabilistic, quantum algorithms are probabilistic in nature; they are designed so that they output the correct answer with high probability (which often means significantly greater than 0 or 1/2). Like all probabilistic algorithms, we may of course amplify the probability of success by repeating the calculation.

One important aspect is what operators are physically realizable; this is expressed by conditions on the matrix A.

Definition 6.2. *A matrix A is unitary if the equation $AA^\star = I$ holds. I is the identity matrix and $A^\star$ is the matrix A both transposed and conjugated.*

Nature forces all quantum operators to be unitary. One way to see that this is a reasonable condition is that being unitary is equivalent to preserving the condition $\sum_\alpha |\varphi(\alpha)|^2 = 1$ for all φ.

Exactly which unitary matrices are allowed depends on the physical realization. We make the bold and rather optimistic assumption that any unitary matrix that only acts on a constant number of qubits can be realized by a physical event. By acting only on a certain set of qubits I we simply mean that $a_{\alpha,\beta}$ only depends on the restriction of α and β to the set I. (Put another way: If α_1 and α_2 are equal on all qubits in I, and β_1 and β_2 are also equal on all qubits in I, then $a_{\alpha_1,\beta_1} = a_{\alpha_2,\beta_2}$.)

It should now be clear that we may not, in general, speed up a computation by combining two operators, A_1 and A_2, into a new operator $A = A_1 A_2$, since A would typically affect too many bits and therefore be physically unrealizable.

The model described above is called the quantum circuit model. We are not concerned with the much more complicated quantum Turing machines in this text. It is also believed that a physical quantum computer would have more resemblance to quantum circuits than to quantum Turing machines. As for deterministic computation, it has been proved that uniform quantum circuits are polynomially equivalent to quantum Turing machines, but the proof is much more complicated [53].

Example 6.3. *Suppose that $m = 1$, i.e., that the quantum system consists of only one qubit. This means that the system can be in two possible states. Let us try to define some operators on this single qubit.*

$$A_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \qquad A_2 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad A_3 = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix}$$

A_1 is the identity matrix and thus corresponds to "no operation."

A_2 is a standard negation, i.e., the values of $\psi(|0\rangle)$ and $\psi(|1\rangle)$ are interchanged.

A_3 might seem to correspond to the operation "write 1," a seemingly useful thing to do. We have a problem with A_3, however, in that the matrix is not unitary. So, this operator is not physically realizable and thus not allowed.

The following Theorem states another important fact about quantum operators.

Theorem 6.4. *Quantum systems are reversible. If A is a valid operator, then A^{-1} is also a valid operator (a unitary matrix that operates on a constant number of qubits.)*

Indirectly, this means that we can not destroy any information: after performing A we can always use the operator A^{-1} to recover the previous state. Since classical deterministic computations are not, in general, reversible, it follows that we cannot directly do normal deterministic calculations. For instance, in Example 6.3 we see that we cannot do a simple operation like writing a constant.

We end this section with an example that illustrates that observing a quantum system in the middle of a computation affects the computation's result.

Example 6.5. *Consider a quantum computer with only one qubit ($m = 1$); the initial state of the system is $|0\rangle$. Let*

$$A = \frac{1}{2}\begin{pmatrix} 1+i & 1-i \\ 1-i & 1+i \end{pmatrix}.$$

Apply the operation A. The system is transformed into the state

$$A\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} \frac{1+i}{2} \\ \frac{1-i}{2} \end{pmatrix}.$$

If we observed the system, what would we see? We would see the states $|0\rangle$ and $|1\rangle$ with the same probability, 1/2 (the squares of the magnitudes of the both components of the vector.)

Now apply A again after having observed the system and read the result. It is not difficult to see that we get a new random bit independent of the first.

This might not seem very interesting, but what would have happened had we applied A twice *before examining the system? The wave function would then have been*

$$AA\begin{pmatrix} 1 \\ 0 \end{pmatrix} = A\begin{pmatrix} \frac{1+i}{2} \\ \frac{1-i}{2} \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix},$$

which means that we always observe $|1\rangle$. Apparently the result may change by observing the system during a calculation.

6.3 Simulating deterministic computations

We can, however, simulate classical deterministic computations by changing the way we compute.

Example 6.6. *We have two bits, x and y, and want to calculate $z = x \wedge y$. This is not immediately possible since we have no way of writing the result to z in a reversible way. To make it reversible we instead update z as $z \leftarrow (x \wedge y) \oplus z$. In other words, results cannot be written directly, but this way we can use them to* change *another bit.*

This is the way the operation affects the bits x, y and z:

$$\begin{aligned} x &\leftarrow x \\ y &\leftarrow y \\ z &\leftarrow z \oplus (x \wedge y) \end{aligned}$$

This most definitely is a reversible operation (just repeating it in fact reverses its effect.) It is easy to conclude that the matrix corresponding to this operation is unitary. It is of dimension 8×8 *since we have 3 bits* $|xyz\rangle$*:*

$$A = \begin{array}{c} \\ |000\rangle \\ |001\rangle \\ |010\rangle \\ |011\rangle \\ |100\rangle \\ |101\rangle \\ |110\rangle \\ |111\rangle \end{array} \begin{array}{c} \begin{array}{cccccccc} |000\rangle & |001\rangle & |010\rangle & |011\rangle & |100\rangle & |101\rangle & |110\rangle & |111\rangle \end{array} \\ \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array}.$$

The gate in the example is referred to as a Toffoli gate. Notice that, if the third input bit z is set to 1, then after applying the Toffoli gate, z is the NAND of x and y. Since NAND gates are universal for deterministic computations, we can in fact simulate all deterministic computations; we describe one way to do it below.

Start with a classical deterministic circuit consisting of only NAND gates that computes $f(x)$ (for which we want to find a quantum circuit.) Replace all gates by Toffoli gates, which are equivalent to the original NAND gates if we supply a number of extra inputs all set to the constant 1.

This implies that if we have an answer of size k and use l gates, we can compute $(x, 0^k, 1^l) \to (x, f(x), y)$, where y are the extra output bits from Toffoli gates. To remove these bits, we proceed as follows.

Copy the result of the computation so that we have $(x, f(x), y, f(x))$. Now reverse the original computation (which is possible since all steps are reversible), to obtain $(x, 0^k, 1^l, f(x))$. The bits that are always the same can be ignored and the theorem below follows.

Theorem 6.7. *If the function* $x \to f(x)$ *can be computed in polynomial time on an ordinary Turing machine (computer), then we can also compute* $x \to (x, f(x))$ *in polynomial time on a quantum computer.*

It might seem like there is no important difference between computing only $f(x)$ and computing $(x, f(x))$. Classically, there is of course not, but the following example shows that for quantum computations such "superfluous" qubits affect the result even if they are not used.

Example 6.8. *Suppose that we are somehow given a single qubit that can be either in the state* $(1/\sqrt{2})|0\rangle + (1/\sqrt{2})|1\rangle$ *or* $(1/\sqrt{2})|0\rangle - (1/\sqrt{2})|1\rangle$*. We want to determine which is the case and apply the operator*

$$A = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

to the bit. (This is in fact the discrete Fourier transform on one qubit.) Depending on the original state of the qubit the resulting state is

$$A \begin{pmatrix} 1/\sqrt{2} \\ 1/\sqrt{2} \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

or

$$A\begin{pmatrix} 1/\sqrt{2} \\ -1/\sqrt{2} \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix},$$

respectively, so measuring the qubit after this operation directly tells us the original state of the qubit.

Now consider what happens if we start out by making a copy of the original bit before investigating its state. (A way to do this would be to apply the identity function on the given qubit in the manner given by Theorem 6.7.)

After copying, the system is in the state $(1/\sqrt{2})|00\rangle+(1/\sqrt{2})|11\rangle$ *or* $(1/\sqrt{2})|00\rangle-(1/\sqrt{2})|11\rangle$. *We perform the same operation as above on the first of the two qubits; the matrix for this operation is*

$$B = \frac{1}{\sqrt{2}} \begin{array}{c} \\ |00\rangle \\ |01\rangle \\ |10\rangle \\ |11\rangle \end{array} \begin{array}{c} \begin{array}{cccc} |00\rangle & |01\rangle & |10\rangle & |11\rangle \end{array} \\ \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix} \end{array}.$$

(The effect on the first qubit is exactly as for the operator A, and the second qubit is left unchanged.) Again, depending on the original state we get

$$B\begin{pmatrix} 1/\sqrt{2} \\ 0 \\ 0 \\ 1/\sqrt{2} \end{pmatrix} = \frac{1}{2}\begin{pmatrix} 1 \\ 1 \\ 1 \\ -1 \end{pmatrix}$$

or

$$B\begin{pmatrix} 1/\sqrt{2} \\ 0 \\ 0 \\ -1/\sqrt{2} \end{pmatrix} = \frac{1}{2}\begin{pmatrix} 1 \\ -1 \\ 1 \\ 1 \end{pmatrix},$$

respectively. Measuring the first qubit after in this case yields $|0\rangle$ *and* $|1\rangle$ *with equal probability and thus does not give any information about the original state of the original qubit. The extra information given by the copy of the original bit prevents the cancellation of amplitudes we had in the first situation.*

If there is an efficient deterministic algorithm that computes the inverse of f, we can get rid of the input from the result to compute just $f(x)$ in the following way.

We apply Theorem 6.7 to f^{-1} to obtain a polynomial time quantum algorithm that makes the computation $f(x) \to (f(x), f^{-1}(f(x)) = (f(x), x)$. Since this computation can be reversed, we can also make the computation $(f(x), x) \to f(x)$. This is exactly the transformation we need.

6.4 Relations to other models of computation

6.4.1 Relation to probabilistic models

The quantum characteristics have similarities to probabilistic computations. The probabilistic operation of writing a single random bit can be represented by the matrix

$$\begin{pmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{pmatrix}$$

in much the same way as we represent quantum operators. The matrix means that, independent of what the value of the single bit, write a 0 with probability $\frac{1}{2}$ or write a 1 with probability $\frac{1}{2}$.

In the same way as with quantum computations, we could for each state α associate a probability p_α of being in that state at a given time.

However, for probabilistic computations it is more efficient to do the random choices "on-line." The reason is that, for classical probabilistic computations, reading the state of the system is possible during the computation without affecting the state.

6.4.2 Simulating quantum computers

Consider a quantum computation on m qubits involving t operations. How fast can we deterministically simulate the output of this quantum computation?

Naively, we would just keep track of 2^m complex numbers representing the wave function of the system, and update this representation for each operation. If the operators are local (i.e., act only on a constant number of qubits), we can calculate each component of the wave function in constant time for each operation since they are determined by a constant number of known values. The time complexity of this approach is $O(t2^m)$, which is very inefficient for large systems.

It is unknown whether one can simulate quantum computers significantly faster than this. The factoring algorithm in the next section presents some evidence that it might not be possible (since it is believed that factoring is hard on a classical computer.) Note that if we could simulate quantum computations efficiently on a classical computer, this would rule out the possibility of solving classically hard problems quickly on a quantum computer.

6.5 The factoring algorithm

This section describes a quantum algorithm for integer factorization. For n bit integer numbers, it requires $O(n)$ qubits and has time complexity $O(n^3)$. For simplicity we omit some details.

To factor the number N, first pick a random number x and compute its order r. The order of x is the smallest r such that $x^r \equiv 1 \pmod{N}$. We hope (this is often true, although we do not prove it here) that (1) r is even and (2) $x^{r/2} \not\equiv -1 \pmod{N}$.

If (1) and (2) hold, the relation $(x^{r/2})^2 \equiv 1^2 \pmod{N}$ holds and computing $\gcd(x^{r/2} - 1, N)$ yields a non-trivial factor.

We need two subroutines:

(1) Pick a random bit The matrix

$$A = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ -\frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \end{pmatrix}$$

converts a 0 to a random bit.

(2) DFT The Discrete Fourier Transform.

The quantum DFT can be described as

$$|a\rangle \rightarrow 2^{-m/2} \sum_{c=0}^{2^m-1} e^{\frac{2\pi i a c}{2^m}} |c\rangle.$$

To understand why this is the Fourier transform of a sequence, we should think of a as representing a vector with a 1 only at position a, and zeros otherwise.

We can also write the quantum DFT operator as a $2^m \times 2^m$ matrix whose entry in row c, column a is $2^{-m/2} e^{\frac{2\pi i a c}{2^m}}$; denote this entry by $A_{c,a}$. It is easy to verify that A is unitary by checking that $A_{c,a} A^\star_{c,a} = 0$ for $a \neq c$ and that $A_{a,a} A^\star_{a,a} = 1$.

However, changing the value of a single input qubit affects all qubits in the output, and it follows that the operator is not local. Thus, we need to factor the matrix into a matrices acting locally. This is accomplished by adapting the FFT (Fast Fourier Transform—see Section ?) algorithm, and it turns out that $O(m^2)$ operators acting on at most 2 qubits is sufficient.

Now, given the input (x, N) where N is an n bit number, we want to compute the order r of x mod N. We also have some extra qubits which are all initially 0. When a bit is not used, we do not write it down when representing the state. The algorithm looks like this:

1. Pick $m = 3n$ random bits to form a random integer a. The state of the system is $\sum_{a=0}^{2^m-1} 2^{-m/2} |a, x, N\rangle$.
2. Compute $x^a \mod N$, which yields the state $\sum_{a=0}^{2^m-1} 2^{-m/2} |x^a, a, x, N\rangle$.
3. Perform the DFT with respect to a. We get
$$2^{-m} \sum_{c=0}^{2^m-1} \sum_{a=0}^{2^m-1} e^{\frac{2\pi i a c}{2^m}} |x^a, c, x, N\rangle.$$
4. Read c.
5. Deterministically find r given c.

Step 2 preserves the input (a, x, N), so this part of the algorithm can be done due to Theorem 6.7, and the implementation of step 3 was discussed above. We need to analyze what properties we can hope of c observed in step 4. The key is to analyze what different a and c in the sum described in step 3 give the same state and thus cause positive or negative interference.

Since c is part of the state, different values of c do not interfere with each other. All a for which x^a have the same value interfere: x^k, x^{k+r}, x^{k+2r}, ... all have the same value (r is the order of x.) Let us look at the corresponding complex numbers. The probability of seeing the state $|x^k, c, x, N\rangle$ can be calculated by taking the squared magnitude of the corresponding complex number which is

$$|2^{-m} \sum_{j=0}^{\lfloor 2^m/r \rfloor} e^{\frac{2\pi i (k+jr)c}{2^m}}|^2.$$

This is not as hard as one might expect since $\sum_{j=0}^{2^m/r} e^{\frac{2\pi i(k+jr)c}{2^m}}$ is a geometric series with first term $s = e^{\frac{2\pi ikc}{2^m}}$ and quotient $k = e^{\frac{2\pi irc}{2^m}}$ between consecutive terms. The well known formula

$$\sum_{i=0}^{n} sk^i = s\frac{1-k^{n+1}}{1-k}$$

is used to yield the result

$$e^{\frac{2\pi ikc}{2^m}} \frac{1 - e^{\frac{2\pi irc(\lfloor 2^m/r \rfloor + 1)}{2^m}}}{1 - e^{\frac{2\pi irc}{2^m}}}.$$

The absolute value of the factor in front of the fraction is 1 and is disregarded. Both the numerator and the denominator of the fraction have magnitudes between 0 and 2. Thus the only way to make this unusually large is by making the denominator close to 0 and the numerator far from 0. That the denominator is close to 0 is equivalent to $e^{\frac{2\pi irc}{2^m}}$ being close to 1 which, in turn, this means that $\frac{rc}{2^m}$ must be very close to an integer. Since we observe c and know 2^m while r is what we want to find it is useful to think of this condition as $c2^{-m}$ being close to a rational number of the form $\frac{d}{r}$. It can be shown (we omit the details) that it is quite likely that we observe a c such that $|\frac{c}{2^m} - \frac{d}{r}| \leq \frac{1}{2^{m+1}}$ holds for some integer d and the key is that this is sufficient to recover r.

We use the continued fraction expansion of $\frac{c}{2^m}$ as discussed in Section ?. As indicated there, the continued fraction expansion of a number Θ gives approximations $\frac{p_i}{q_i}$ such that

$$|\Theta - \frac{p_i}{q_i}| \leq \frac{1}{q_i^2},$$

but there is also a converse to this: any approximation such that

$$|\Theta - \frac{p}{q}| \leq \frac{1}{2q^2}$$

is in fact one of the approximations output by the continued fraction algorithm. In particular, since $2r^2 \leq 2^{m+1}$ the approximation d/r to $c/2^m$ is found.

When we finally have a candidate for r, it can be used in the equation $\left(x^{r/2}\right)^2 = 1^2 \pmod N$ as described in the beginning of this section. If we are unsuccessful in finding a non-trivial factor, we try again with a new x. For missing details in the description and the analysis we refer to [49].

6.6 Are quantum computers for real?

Some physicists claim that it is impossible to build large scale quantum computers that work for a large number of steps. One of the largest problems is that quantum systems tend to interact with their environment so that the state of the system is lost very quickly. The most sophisticated experimental setups have been able to implement a few very simple operations on very few qubits (less than 10).

The factoring algorithm described in the previous section example would of course require thousands of qubits for many thousands of time steps to factor a numbers intractable by classical methods. One direction of research trying to overcome the difficulties of quantum decoherence is to integrate error correction techniques into quantum algorithms (see [50]).

The accuracy needed for the operators used by the algorithms may also prove very hard to achieve physically.

Another area of quantum information processing under development is quantum cryptography. Experimental quantum cryptographic systems have already been built, and it seems quite likely that such systems will work well in practise long before quantum computers do. Ironically, since there are classical cryptographic systems that rely on the hardness of integer factorization, there is no immediate need for quantum cryptography as long as factorization remains a difficult task, which might be as long as there are no large scale quantum computers.

Chapter 7

Factoring polynomials in finite fields

An algorithm for factoring polynomials is a basic component in many computer algebra packages. It may be surprising but true that factoring polynomials is much easier than factoring integers and we start by factoring polynomials modulo small primes p.

Assume that we are given a polynomial $g(x)$ of degree n and we want to decide if is irreducible or factor it into irreducible factors. An easy case to handle is when $g(x)$ contains a square factor, *i. e.* a factor of the form $p(x)^2$. In this case $p(x)$ divides $gcd(g(x), g'(x))$ where g' is the derivative of g. Over a discrete domain the derivative does not have the same "slope" interpretation as over the real numbers and we simply form it in a syntactic manner by defining the derivative of x^d to be dx^{d-1} and extending it by linearity. It is not difficult to check that all the usual theorems apply and in particular the derivative of $p(x)^2h(x)$ is $p(x)^2h'(x) + 2p'(x)p(x)h(x)$. Thus the factor $p(x)$ can be found efficiently using the Euclidean algorithm applied to polynomials.

Note that over a finite field $g'(x)$ might be identically zero even if $g(x)$ is not a constant. This happens mod p when $g(x) = h(x^p)$ for some polynomial h. In this case $GCD(g(x), g'(x)) = g(x)$ so we do not get a nontrivial factorization. However, mod p it is true [1] that $h(x^p) = h(x)^p$ and thus we simply proceed to factor h. Let us now concentrate on the case of $p = 2$ and we later discuss how to extend the results to general p. Keep in mind that all arithmetic from now on is done mod 2.

The key to our algorithm is the mapping

$$f(x) \mapsto f(x)^2 \mod g(x)$$

of polynomials of degree at most $n - 1$. This is a linear map since $(f_1(x) + f_2(x))^2 = f_1(x)^2 + f_2(x)^2$ whenever we are in characteristic 2. We are interested in the fix-points of this map, *i. e.* polynomials $f(x)$ such that

$$f(x) = f(x)^2 \mod g(x) \tag{7.1}$$

[1] This is established by making the multinomial expansion of $h(x)^p$. All multinomial coefficients except those appearing in front of the terms in $h(x^p)$ are divisible by p and hence 0 mod p.

If g is irreducible, then working mod $g(x)$ is actually working in the finite field $GF[2^n]$ and in any field a quadratic equation has at most 2 solutions and it is easy to see that the given equation has exactly two solutions namely the constants 0 and 1.

If g is the product of a number of irreducible polynomials (let us assume two for concreteness), the situation changes. If $g(x) = g_1(x)g_2(x)$ then by the Chinese remainder theorem, each polynomial f mod g can instead be represented as the pair (f_1, f_2) where $f(x) = f_i(x) \mod g_i(x)$. Then f solves equation (7.1) iff f_1 solves it mod g_1 and f_2 solves it mod g_2. Since we know the set of solutions mod irreducible polynomial, this implies that we get 4 solutions mod $g(x)$. These are (when written as pairs, *i. e.* the first component is the remainder mod g_1 and the second mod g_2) $(0,0),(1,1),(0,1)$ and $(1,0)$. The first two solutions correspond to the standard roots $f(x)$ being the constants 0 or 1 while the other two are much more interesting. If $f_{(0,1)}(x)$ is the polynomial corresponding to the third solution then $GCD(g(x), f_{(0,1)}(x)) = g_1(x)$ and our problem is solved. In general the number of fix-points is given by 2^l where l is the number of irreducible factors in g. Let us see how this works in a couple of examples.

Example 7.1. *Let $g(x) = x^5 + x^2 + 1$. Then under our mapping*

$$\begin{aligned} 1 &\mapsto 1 \\ x &\mapsto x^2 \\ x^2 &\mapsto x^4 \\ x^3 &\mapsto x^6 = x^3 + x \\ x^4 &\mapsto x^8 = x^2 \cdot x^6 = x^5 + x^3 = x^3 + x^2 + 1 \end{aligned}$$

Thus in the basis 1, x, x^2, x^3, x^4 our mapping is given by the matrix.

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

Since we are looking for fix-points to the mapping given by M we want to find the null-space of the mapping given by the matrix $M + I$ where I is the identity matrix i. e. *we want to study the matrix.*

$$M + I = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

It is easy to check that the null-space of this matrix is given by the only vector $(1,0,0,0,0)$ and thus $x^5 + x^2 + 1$ is irreducible.

Example 7.2. *Let $g(x) = x^5 + x + 1$. This time*

$$\begin{array}{rcl} 1 & \mapsto & 1 \\ x & \mapsto & x^2 \\ x^2 & \mapsto & x^4 \\ x^3 & \mapsto & x^6 = x^2 + x \\ x^4 & \mapsto & x^8 = x^2 \cdot x^6 = x^4 + x^3 \end{array}$$

$$M = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

which gives

$$M + I = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix}$$

The null-space of this matrix is spanned by $(1, 0, 0, 0, 0)$ and $(0, 1, 0, 1, 1)$. The first is the trivial solution, while the second corresponds to the polynomial $x^4 + x^3 + x$. Computing $GCD(x^5 + x + 1, x^4 + x^3 + x)$ by the Euclidean algorithm gives

$$\begin{array}{rcl} x^5 + x + 1 & = & (x + 1)(x^4 + x^3 + x) + x^3 + x^2 + 1 \\ x^4 + x^3 + x & = & x(x^3 + x^2 + 1) \end{array}$$

Finally a division gives $x^5 + x + 1 = (x^3 + x^2 + 1)(x^2 + x + 1)$ which is the complete factorization.

Finding the nullspace is just linear algebra and can be done by Gaussian elimination. Since the *gcd* can be computed even faster we get.

Theorem 7.3. *We can factor degree n polynomials over $GF[2]$ into irreducible factors in time $O(n^3)$.*

We have in fact only established how to factor a non-irreducible polynomial into two factors. We leave the details how to get the complete factorization within the same time-bound as an exercise. We note, however, that the computation of the nullspace need not be redone and thus the hard part of the computation can be reused.

7.1 Factoring polynomials in larger fields

Let us see what happens for primes p other than 2. Much of the work transfers without change and in particular we should study the mapping

$$f(x) \mapsto f(x)^p = f(x^p) \bmod g(x)$$

which is again a linear map. The set of fix-points to this map is a linear subspace of the polynomials of degree at most $n-1$ of dimension l where l is the number of irreducible factors in g. The fix-points are all polynomials $h(x)$ such that h is a constant (degree 0 polynomial) mod g_i for each irreducible factor g_i of g. If g is not irreducible, then, in the mod 2 case, it was true that for any non-constant fix-point h, $gcd(h, g)$ was nontrivial. This is not true for larger p, *e. g.* for $p = 3$ we can have $g(x) = g_1(x)g_2(x)$ and $h(x) = 1 \bmod g_1(x)$ and $h(x) = 2 \bmod g_2(x)$, then h is a fix-point while $gcd(g, h) = 1$. However, it is always true that for some i, $0 \le i \le p-2$, $gcd(h+i, g)$ is nontrivial. Thus $p-1$ gcd computations is always sufficient to find a nontrivial factor.

Let us analyze the running time of this algorithm. For simplicity let us count the number of operations that are done on integers mod p rather than bit-operations. Let us first assume that $p \le n$. Then using $x^{p(i+1)} = x^p(x^{pi})$ it is easy to see that each column in the matrix M can be computed from the previous in $O(pn)$ operations. To determine the null-space can be done by Gaussian elimination in $O(n^3)$ operations and finally each GCD can be done in $O(n^2)$ operations giving a total of $O(pn^2 + n^3) = O(n^3)$ operations on integers mod p.

If p is large then the cost $O(pn^2)$ becomes the dominant cost. However in the first step (constructing the matrix M) we can be more efficient. We can precompute $x^p \bmod g(x)$ by $O(\log p)$ squarings and multiplications[2] by x and this can be done in $O(n^2 \log p)$ operations. Each successive column can then be computed $O(n^2)$ operations. Thus the first part of the algorithm that determines the number of irreducible factors in $g(x)$ (and in particular if g is irreducible) can be made to run in $O(n^2 \log p + n^3)$ operations. However, the second step (factorization of non-irreducible g) is inefficient for large p.

To get an efficient algorithm for large p we need to make an additional observation. The inefficiency comes from the fact that if h is a fix-point, then we know that $h(x) \bmod g_1(x)$ is one of the numbers $0, 1, \ldots p-1$ but we do not know which. We need to narrow the choice. The trick is to consider $h(x)^{(p-1)/2}$. Since for any $a \ne 0 \bmod p$ we have $a^{(p-1)/2} = \pm 1$ (half of the a's giving each possibility) we are in much better shape. The second step is thus transformed to

- For a random fix-point h compute $GCD(h^{(p-1)/2} - 1, g(x))$.

The polynomial $h^{(p-1)/2} \bmod g(x)$ can be computing in $O(n^2 \log p)$ operations by the method of repeated squaring. It is possible to show that with probability at least one half the above GCD is nontrivial. Thus we get a method that runs in expected time $O(n^2 \log p + n^3)$. Note that the answer we get is always correct since the part determining the number of irreducible factors is deterministic. We sum up the conclusions in a theorem.

Theorem 7.4. *Factorization of polynomials in $GF[p]$ can be done in $O(n^3 + n^2 \log p)$ time. Time is counted as the number of field operations. The algorithms is probabilistic and the running time is only expected time. The answer, however, is always correct.*

[2] Compare to how we computed $a^{p-1} \bmod p$ when we were doing primality tests.

7.2 Special case of square roots

Let us consider the problem of computing square roots modulo large numbers p. This is a special case of polynomial factorization since taking the square root of a is equivalent to factoring $x^2 - a$. Let us first check that our test for irreducibility works out. With $g(x) = x^2 - 1$ we have a degree polynomial and we just need to find out to what values 1 and x maps under $f(x) \mapsto f(x)^p$. Clearly 1 goes to 1 and x goes to $x^p = xx^{2((p-1)/2)} = xa^{(p-1)/2}$. Thus the matrix whose nullspace we should find is given by:

$$\begin{pmatrix} 0 & 0 \\ 0 & a^{(p-1)/2} - 1 \end{pmatrix}$$

and this has null-space dimension 2 iff $a^{(p-1)/2} = 1$ which is just the standard criteria for a being a square mod p. So far so good!

Now let us try to find the factors. According to our general algorithm we should take a random polynomial h and then compute $GCD(h(x)^{(p-1)/2} - 1, x^2 - a)$. Since h should be a polynomial from the null-space which is not a constant the first value that comes to mind is $h(x) = x$. Let us try this. We need to consider two cases, when $p = 1 \bmod 4$ and $p = 3 \bmod 4$. Let us start with the second. In this case $x^{(p-1)/2} = xx^{2((p-3)/4)} = xa^{(p-3)/4}$. Thus $x^{(p-1)/2} - 1 = xa^{(p-3)/4} - 1$ and it is easy to check that

$$x^2 - a = (xa^{(p-3)/4} - 1)(xa^{(p+1)/4} + a)$$

which implies that we get a solution $x = a^{(3-p)/4} =$ (since $a^{(p-1)/2} = 1$) $a^{(p+1)/4}$. That the solutions is correct can be verified directly since

$$(a^{(p+1)/4})^2 = aa^{(p-1)/2} = a,$$

and we state this as a theorem.

Theorem 7.5. *It is possible to extract square-roots mod p where p is a prime which is 3 mod 4 in deterministic time $O((\log p)^3)$ (counted as bit operations).*

Example 7.6. *Assume that $p = 31$ and let us extract the square root of 18. We know from above that this number is simply 18^8 and thus we simply compute*

$$18^8 \equiv (324)^4 \equiv 14^4 \equiv (196)^2 \equiv 10^2 \equiv 7$$

and in fact $7^2 = 49 \equiv 18$. The other square root of 18 is of course $-7 \equiv 24$.

Now let us turn to the case when $p = 1 \bmod 4$. Then

$$x^{(p-1)/2} = x^{2((p-1)/4)} = a^{(p-1)/4}.$$

This is just a constant (in fact either 1 or -1) and this implies that $GCD(x^{(p-1)/2} - 1, x^2 - a)$ is never a nontrivial factor. Thus in this case we have to try another $h(x)$. It turns out that in the case $p = 1 \bmod 4$, there is no known algorithm for extracting square roots that run in deterministic polynomial time. We have to resort to the general procedure of picking a random h.

Theorem 7.7. *It is possible to extract square-roots modulo p where $p \equiv 1 \bmod 4$ in probabilistic time $O((\log p)^3)$ (counted as bit operations). The answer is always correct.*

Example 7.8. *Let us extract the square root of 23 modulo 29. We do this by computing picking* $h(x) = x + 1$ *and computing* $h'(x) = h(x)^{14} \bmod x^2 - 23$

$$
\begin{aligned}
h(x)^2 &= x^2 + 2x + 1 \equiv 2x + 24 \\
h(x)^3 &= (x+1)(2x+24) = 2x^2 + 26x + 24 \equiv 26x + 12 \\
h(x)^6 &= (26x+12)^2 = 676x^2 + 624x + 144 \equiv 15x + 3 \\
h(x)^7 &= (x+1)(15x+3) = 15x^2 + 18x + 3 \equiv 18x \\
h(x)^{14} &= (18x)^2 = 324x^2 \equiv 28
\end{aligned}
$$

and thus we do not get an nontrivial factor for this choice of h*. We instead set* $h(x) = x + 2$ *getting*

$$
\begin{aligned}
h(x)^2 &= x^2 + 4x + 4 \equiv 4x + 27 \\
h(x)^3 &= (x+2)(4x+27) = 4x^2 + 35x + 54 \equiv 6x + 1 \\
h(x)^6 &= (6x+1)^2 = 36x^2 + 12x + 1 \equiv 12x + 17 \\
h(x)^7 &= (x+1)(12x+17) = 12x^2 + 41x + 34 \equiv 12x + 20 \\
h(x)^{14} &= (12x+20)^2 = 144x^2 + 480x + 400 \equiv 16x.
\end{aligned}
$$

In this case $gcd(16x - 1, x^2 - 23)$ *is simply* $16x - 1$*, or by multiplying by the constant 20,* $x - 20$ *and thus we have that one square root of 23 is 20, the other square root being* $-20 \equiv 9$*.*

Thus we have a big difference for extracting square roots depending on p modulo 4. If one thinks about it for a while it turns out that the reason for the difference is -1 is a quadratic non-residue modulo p iff $p \equiv 3 \bmod 4$, and -1 is exactly the factor we need to multiply one square root by to get the other. We invite the interested reader to clarify this remark.

An even bigger difference is given by extracting square roots modulo composite numbers. This in fact turns out to be as hard as factoring. To see this, note that some of our factoring algorithms depended on our ability to find x and y such that $x \not\equiv \pm y$ and $x^2 \equiv y^2$. If we could extract square roots then such a pair could be constructed by simple picking a random x computing $a \equiv x^2$ and then extracting the square root of a. This would give an y that always satisfies $y^2 \equiv x^2$ and it is not hard to see that $y \not\equiv \pm x$ with probability as least $1/2$.

Conversely, if we can factor we can also extract square roots. This follows from the fact that we can extra square roots modulo the prime factors and then combine the results by efficient Chinese remaindering.

Chapter 8

Factoring polynomials over integers

We now turn to the factorization of ordinary integer polynomials. using ordinary arithmetic. We are thus given $g(x) = \sum_{i=0}^{n} g_i x^i$, where $g_i \in \mathbb{Z}$. To begin with we have to consider which domain to allow for the coefficients. The most general domain would be the complex numbers but this is simply root-finding since

$$g(x) = \prod_{i=1}^{n} (x - x_i)$$

where x_i are the roots to $g(x) = 0$. Root-finding is an important problem but we do not discuss it here. Since g is a real polynomial the roots comes as real roots and pairs of conjugated roots and thus the factorization of g over the real numbers is given by

$$\prod_{x_i \text{ real}} (x - x_i) \prod_{(x_i, \overline{x}_i) \text{ conjugate pair}} (x - x_i)(x - \overline{x}_i)$$

since $(x - x_i)(x - \overline{x}_i)$ is a real polynomial of degree 2. Thus the interesting domain for factorization of polynomials with integer coefficients is the domain of rational numbers. We can clearly assume that the gcd of the coefficients of g is 1 since otherwise we can simply divide all coefficients by their common gcd. Furthermore, if the factorization of g is $h_1 h_2$ then, by multiplying h_1 by suitable rational number and dividing h_2 by the same number, we can assume that also h_1 have integer coefficients with gcd 1. In this situation it turns out (see Lemma 8.1 below) that h_2 also has integer coefficients and in fact we need not leave the domain of integers.

Lemma 8.1. *Let g and h_1 be integer polynomials such that the greatest common divisor of the coefficients of either polynomial is 1. Then if $g(x) = h_1(x)h_2(x)$ where $h_2(x)$ is a polynomial with rational coefficients, the coefficients of h_2 are in fact integral.*

Proof. For contradiction, assume h_2 does not have integer coefficients and that the least common denominator of the coefficients in h_2 is M. Let p be a prime factor in M and consider the following equation.

$$M \cdot g(x) = h_1(x) \cdot M h_2(x)$$

$Mh_2(x)$ is an integer polynomial. If we look at this equation modulo p, $Mg(x) \equiv 0$, but $h_1(x) \not\equiv 0$ since the greatest common divisor of the coefficients of h_1 is 1. This also applies for Mh_2, so we have reached a contradiction. Here, we need the fact that the product of two nonzero polynomials modulo p is a nonzero polynomial. This can be seen by considering the leading terms in the three polynomials. □

Another question is how large the coefficients of h_1 and h_2 can be. They can be considerably larger than the coefficients in g! We can estimate how large the coefficients of a polynomial can be in terms of the size of the complex roots of the polynomial. Let $M = \max\{|g_0|, \ldots, |g_n|\}$ and write g as $g(x) = g_n \prod_{i=1}^{n}(x - x_i)$ where x_i is the complex roots of $g(x) = 0$.

Let us first estimate the magnitude of the roots.

Lemma 8.2. $|x_i| \leq 2M/|g_n|$

Proof. Assume that $|x| > 2M/|g_n|$ and we want to prove that $g(x)$ is nonzero. The idea is that in this case $g_n x^n$ is the dominant term and cannot be cancelled.

$$|g(x)| = \left|\sum_{i=0}^{n} g_i x^i\right| = |g_n x^n| - \left|\sum_{i=0}^{n-1} g_i x^i\right| \geq |g_n||x^n| - M \sum_{i=0}^{n-1} |x|^i$$

The sum in the last term is a geometric series with quotient ≥ 2 and hence the sum is at most twice the last term. Therefore,

$$|g_n||x^n| - M \sum_{i=0}^{n-1} |x|^i > |g_n||x^n| - 2M|x|^{n-1} \geq |x|^{n-1}(|g_n||x| - 2M) \geq 0$$

and since we had strict inequality in one of the steps $g(x)$ is nonzero. □

We can use this estimate to give an upper bound on the sizes of the coefficients of the factors of g. Let $h_1(x) = h_1^{lead} \prod_{i \in S}(x - x_i)$ where h_1^{lead} is the leading coefficient in h_1 and S is the set of roots of g that also solve $h_1(x) = 0$. We know that h_1^{lead} divides g_n, so $|h_1^{lead}| \leq |g_n|$.

If h_1 has degree m then the coefficient for degree i is bounded by

$$h_1^{lead} \binom{m}{i} \left(\frac{2M}{|g_n|}\right)^{m-i} \leq |g_n| \cdot 2^m \cdot \left(\frac{2M}{|g_n|}\right)^m \leq (4M)^m \leq (4M)^n.$$

Thus we have proved the following lemma.

Lemma 8.3. *Let g be an integer polynomial of degree n such that any coefficient is bounded by M in absolute value. Then any integer factor h of g has coefficients bounded by $(4M)^n$.*

After these preliminaries, let us now turn to the actual problem of factorization. Since we already know how to factor in finite fields and any factorization over the integers immediately gives a factorization modulo p, for any p, it is natural to try to use these factorization in some way. One idea would be to factor modulo the upper bound we have. In other words, to do the factorization modulo a prime $p > (4M)^n$ and then test if the factorization actually comes from a factorization over the integers. If we would obtain a factorization modulo

p with only two factors then we are done since we just check if this is actually a factorization over the integers[1] and if this is not the case then g is irreducible.

If g is the product of many factors modulo p, say even that g splits into linear factors $g(x) = \prod_{i=1}^{n} g_i(x) \bmod p$ then we are in trouble. The problem being that a factor of the integers might be a product of many factors modulo p and thus we need to try the possible factors $h(x) = \prod_{i \in S} q_i(x) \bmod p$ for any subset S and in the worst case there are 2^n possible S. If this happens we could simply try a different p and hope for fewer factors. This approach does work well for a random polynomial and it is the approach we would recommend in practice. There are bad examples, however, with many factors modulo p for any p but no factors over the integers and to be theoretically accurate we now turn to an algorithm that works in polynomial time for all polynomials. It is probably a bit slow in practice.

8.1 The algorithm

A high level description of our algorithm is as follows.

1. Find factors $g(x) = h_1(x)h_2(x)$ modulo p^k for some p and k. Think of p as small and k as large. Here h_1 is an irreducible polynomial and h_2 is "the rest".

2. Find a polynomial $h(x) = h_1(x)z(x) \bmod p^k$ where $deg(h) < deg(g)$. The coefficients of h should be small, say less than M_1.

3. Compute $gcd(g(x), h(x))$.

We want the gcd to be non-trivial and a simple condition for this property is useful. Assume that $deg(f) = n$ and $deg(g) = m$.

Lemma 8.4. *$gcd(f(x), g(x))$ is non-trivial if and only if there is a non-trivial solution to $f(x)h_1(x) + g(x)h_2(x) = 0$ with $deg(h_1) < m$ and $deg(h_2) < n$.*

Proof. Say $gcd(f(x), g(x)) = d(x)$. Then $h_1(x) = g(x)/d(x)$ and $h_2(x) = -f(x)/d(x)$ satisfy the equation. Conversely, if $gcd(f(x), g(x)) = 1$ then every irreducible factor in g divides $h_1(x)$ which in turn means that $g(x)|h_1(x)$. This is a contradiction because $deg(h_1) < deg(g)$. □

The condition of Lemma 8.4 is a matter of solvability of a homogeneous linear equation system and we can check this efficiently by looking the appropriate matrix and its determinant. We call it the *Sylvester resultant.*

[1] We are actually cheating slightly since in the modulo p factorization we only know the factors upto an arbitrary constant multiplier and we need to find this multiplier. This is not difficult in practice, but we omit the discussion how this is done.

8.1.1 The Sylvester resultant

Define the matrix S as

$$\underbrace{\begin{pmatrix} f_0 & f_1 & f_2 & \dots & f_{n-1} & f_n & 0 & \dots & 0 \\ 0 & f_0 & f_1 & \dots & f_{n-2} & f_{n-1} & f_n & \dots & 0 \\ \vdots & & & & & & & & \\ 0 & 0 & 0 & \dots & f_0 & f_1 & f_2 & \dots & f_n \\ g_0 & g_1 & g_2 & \dots & g_m & 0 & 0 & \dots & 0 \\ 0 & g_0 & g_1 & \dots & g_{m-1} & g_m & 0 & \dots & 0 \\ \vdots & & & & & & & & \\ 0 & 0 & 0 & \dots & g_0 & g_1 & g_2 & \dots & g_m \end{pmatrix}}_{n+m \text{ elements}}$$

What we want to check is whether the system

$$S^T \begin{pmatrix} h_1^{(0)} \\ h_1^{(1)} \\ \vdots \\ h_1^{(m-1)} \\ h_2^{(0)} \\ h_2^{(1)} \\ \vdots \\ h_2^{(n-1)} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

is solvable. The Sylvester resultant $R(f, g)$ is defined as the determinant of S and the system is solvable by a non-trivial solution if $R(f, g) = 0$. So by the above lemma, $gcd(f(x), g(x))$ is non-trivial if and only if $R(f, g) = 0$ and this applies both over the integers, modulo p and modulo p^k.

Example 8.5. *We are given polynomials $f(x) = 1+2x+x^3$ and $g(x) = 5+x^2$. Does there exist polynomials $h_1(x) = h_1^0 + h_1^1 x$ and $h_2(x) = h_2^0 + h_2^1 x + h_2^2 x^2$ such that $h_1(x)f(x) + h_2(x)g(x) = 0$? We get the linear system*

$$\begin{pmatrix} 1 & 0 & 5 & 0 & 0 \\ 2 & 1 & 0 & 5 & 0 \\ 0 & 2 & 1 & 0 & 5 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} h_1^0 \\ h_1^1 \\ h_2^0 \\ h_2^1 \\ h_2^2 \end{pmatrix} = \bar{0}$$

which is solvable if

$$R(f, g) = \begin{vmatrix} 1 & 2 & 0 & 1 & 0 \\ 0 & 1 & 2 & 0 & 1 \\ 5 & 0 & 1 & 0 & 0 \\ 0 & 5 & 0 & 1 & 0 \\ 0 & 0 & 5 & 0 & 1 \end{vmatrix} = 0.$$

In this case, $R(f, g) = 46$ and the system has therefore no solution except the zero vector. It does have a solution modulo 23 where we can set $h_1(x) = 15 - x$ and $h_2(x) = -3 + 8x + x^2$.

We can now see that the outlined algorithm should work, because with $h_1(x)$ dividing both $h(x)$ and $g(x)$ modulo p^k, the Sylvester resultant equals 0 modulo p^k since $gcd(h(x), g(x))$ modulo p^k is non-trivial. We want that $R(h, g) = 0 \bmod p^k$ together with the fact the coefficients of h are small would imply that $R(h, g) = 0$ over the integers. This is in fact not hard to achieve and we state it as a lemma.

Lemma 8.6. *Let h_1 be an integer polynomial of degree m with coefficients bounded by M_1 and let g be an integer polynomial of degree n with coefficients bounded by M. If $gcd(h_1, g)$ is nontrivial modulo p^k then, provided $p^k > (n + m)!M^m M_1^n$, $gcd(h_1, g)$ is nontrivial also over the integers.*

Proof. Look at $R(h_1, g)$. This number is 0 modulo p^k and we need to prove that it is 0 also over the integers since this is equivalent with the conclusion of the lemma. Now $R(h_1, g)$ is a determinant of size $n + m$ and the first m rows have elements bounded by M while the last n rows have elements bounded by M_1. The expansion defining the determinant contains $(m + n)!$ terms where each is a product of $(n + m)$ factors, one from each row. Thus

$$|R(h_1, g)| \leq (n + m)!M^m M_1^n$$

and thus by the condition on the size of p^k and the fact that it is 0 modulo p^k we can conclude that it is 0 over the integers. □

8.1.2 Finding a factorization of $g(x)$ modulo p^k

The first task in the algorithm is to find factors $h_1(x)$ and $h_2(x)$ such that $g(x) = h_1(x)h_2(x)$ modulo some p^k. We can proceed as follows.

(i) Choose p such that $g(x)$ is square free modulo p.

(ii) Factor $g(x)$ modulo p.

(iii) For $i := 1$ to $k - 1$, "lift" the factorization modulo p^i to modulo p^{i+1}.

We know from the last chapter how to do item (ii). Item (iii) is simplified by having $g(x)$ square free, which explains why we do item (i). Having $g(x)$ square free is equivalent to $gcd(g, g')$ being trivial, which in it's turn is equivalent to $R(g, g') \neq 0$, so we want to choose a p that does not divide $R(g, g')$. We can use the fact that $R(g, g') \leq M^{2n}(2n)!$ to find such a prime number. If the resultant would be divisible by every prime up to some x, then $R(g, g')$ would be in the order of e^x, so there must be a $p < \log(M^{2n}(2n)!) = 2n \log M + 2n \log n$ we can use, and we can find it by simply computing the resultant over the integers and then trying all small primes. Note that we can assume $g(x)$ to be square free over the integers, because if that was not the case, we can easily get a factor of $g(x)$ by computing $gcd(g, g')$.

The task (iii) can be done with a procedure called *Hensel lifting*. We have $g(x) = h_1^{(i)}(x)h_2^{(i)}(x)$ modulo p^i and want $g(x) = h_1^{(i+1)}(x)h_2^{(i+1)}(x)$ modulo p^{i+1}. Try $h_1^{(i+1)}(x) = h_1^{(i)}(x) + p^i h_1'(x)$ and $h_2^{(i+1)}(x) = h_2^{(i)}(x) + p^i h_2'(x)$. Then

$$h_1^{(i+1)}(x)h_2^{(i+1)}(x) = h_1^{(i)}(x)h_2^{(i)}(x) + h_1^{(i)}(x)p^i h_2'(x) + h_2^{(i)}(x)p^i h_1'(x) + p^{2i} h_1'(x)h_2'(x)$$

would hopefully equal $g(x)$ modulo p^{i+1}. This can be written as the equation

$$p^i g'(x) = h_1^{(i)}(x)p^i h_2'(x) + h_2^{(i)}(x)p^i h_1'(x) \bmod p^{i+1}$$

if we let $p^i g'(x) = g(x) - h_1^{(i)}(x)h_2^{(i)}(x)$. Then we can divide by p^i everywhere and get

$$h_1^{(i)}(x)h_2'(x) + h_1'(x)h_2^{(i)}(x) = g'(x) \bmod p$$

This is a linear equation system modulo p and if it is solvable, we simply find the solution by Gaussian elimination. It turns out that this system is always solvable when $g(x)$ is square free. We omit the details. There are $n+2$ unknowns in h_1' and h_2' so we can fix one parameter, say the leading coefficient of h_1', and solve.

The whole procedure of finding a factorization of $g(x)$ modulo p^k involves k systems of linear equations modulo p of size $n+1 \times n+1$.

8.1.3 Finding a small polynomial with a factor in common with $g(x)$

We now proceed with the second task of the algorithm, *i. e.* to find a polynomial $h(x) = h_1(x)z(x)$ modulo p^k, $deg(h) \leq n-1$, with small coefficients. The tool here is Lovász' algorithm for finding a short vector in a lattice. Given a set of vectors $(\vec{b}_i)_{i=1}^n$, the corresponding lattice is defined as all integer combinations of these vectors *i. e.* all vectors of the form $\sum_{i=1}^n a_i\vec{b}_i, a_i \in \mathbb{Z}$. To use this notation, we view our polynomials as vectors.

A polynomial $f(x) = \sum_{i=0}^m f_i x^i$ has a corresponding vector $(f_0, f_1, \ldots, f_m, 0, 0, \ldots, 0)$ in an n-dimensional space. Some consequences:

$$\begin{array}{rcl} xf & \leftrightarrow & (0, f_0, f_1, \ldots) \\ x^{n-m-1}f & \leftrightarrow & (0, 0, \ldots, 0, f_0, \ldots, f_m) \end{array}$$

In our application, L_{h_1} is the lattice spanned by the vectors corresponding to $x^i h_1, 0 \leq i \leq n-m-1$ and $p^k\vec{e}_i$, where $\vec{e}_i$ are the unit vectors. The vectors in this lattice corresponds exactly to the multiples of h modulo p^k. If we can find a short vector in L_{h_1}, then we have found the polynomial h with small coefficients.

Lovász' algorithm finds a vector of length less than $2^{(n-1)/2}\lambda$ in a lattice where λ is the length of the shortest vector. With input vectors of length B, the running time is bounded by $O(n^5 \log B)$ arithmetic operations. The details of that algorithm is given in Chapter 9

In our case we know that if g is not irreducible, then, by Lemma 8.3, there is a vector of length at most $\sqrt{n}(4M)^n$ in the lattice. The length of the output vector is thus at most $2^{n/2}\sqrt{n}(4M)^n \leq (8M)^n$. This output vector can be translated into a polynomial h that is a multiple of h_1, and with coefficients bounded by $(8M)^n$. Thus if we choose $p^k > (n+m)!M^m M_1^n \sim (8M)^{n^2}$ we are guaranteed that $gcd(g, h)$ is non-trivial and this greatest common divisor can of course be calculated efficiently.

The length of the input vectors is in the order of $p^k \sim (8M)^{n^2}$. Therefore, the running time of Lovász' algorithm, which also dominates the whole factorization, comes to $O(n^5 \cdot \log((8M)^{n^2})) = O(n^7 \log M)$. This running time can be improved with a more careful implementation and analysis.

We state the last conclusion as a theorem

Theorem 8.7. *It is possible to factor polynomials of degree n over the integers with coefficients bounded by M in $O(n^7 \log M)$ arithmetic operations.*

Chapter 9

Lovász's Lattice Basis Reduction Algorithm

Factorization of integer polynomials required a subroutine for finding a short vector in a lattice. This subroutine is very useful and has been applied in a number of contexts, the most publicized possibly being breaking [8, 48] a number of the public key cryptosystems based on knapsacks. We try to give most details of this algorithm.

9.1 Definition of lattices and some initial ideas

Given vectors $\vec{b}_1, \vec{b}_2, \ldots, \vec{b}_n$, we wish to study all vectors in the set

$$L = \{\sum_{i=1}^{n} a_i \vec{b}_i \mid a_i \in \mathbb{Z}\},$$

and find one that isn't too long. If the $\vec{b}_i$ form a basis of $\mathbb{R}^n$, $L \subset \mathbb{R}^n$ is called a *lattice*. We assume for simplicity that $\vec{b}_1, \vec{b}_2, \ldots, \vec{b}_n \in \mathbb{Z}^n$ are linearly independent and hence form a basis of $\mathbb{R}^n$.

Example 9.1. *If $n = 2, \vec{b}_1 = (1,0), \vec{b}_2 = (0,1)$, then $L = \{(a,b) \mid a, b \in \mathbb{Z}\}$, which is the set of all integral points in the plane.*

Example 9.2. *If we instead let $\vec{b}_1 = (2,0)$, then L is all integral points whose first coordinate is even.*

Example 9.3. *If we let $\vec{b}_1 = (1,2)$ and $\vec{b}_2 = (-1,3)$ then L consists of all integral points such that the inner product with the vector $(3,1)$ is divisible by 5. Note that the two vectors $\vec{b}_1' = (0,5)$ and $\vec{b}_2 = (1,2)$ span the same lattice.*

Thus a lattice may have many bases and a major problem is choosing the "best" basis. Let us give a more elaborate example illustrating this point.

Example 9.4. *Let $n = 2, \vec{b}_1 = (421,114)$ and $\vec{b}_2 = (469,127)$. Now suppose we change this basis by keeping $\vec{b}_1$ but changing $\vec{b}_2$ to $\vec{b}_2' = \vec{b}_2 - \vec{b}_1 = (48,13)$.*

Clearly we generate the same lattice this way but the new basis is better in that at least one of the vectors is shorter. We can continue this process. Set

$$\vec{b}_1' = \vec{b}_1 - 9\vec{b}_2 = (-11, -3)$$

now $\vec{b}_1'$ *and* $\vec{b}_2'$ *is an even better basis. We can continue setting*

$$\begin{aligned} \vec{b}_2'' &\leftarrow \vec{b}_2' + 4\vec{b}_1' = (4,1) \\ \vec{b}_1'' &\leftarrow \vec{b}_1' + 3\vec{b}_2'' = (1,0) \\ \vec{b}_2''' &\leftarrow \vec{b}_2'' - 4\vec{b}_1'' = (0,1). \end{aligned}$$

with each pair generating the same lattice. Thus in other words the basis $(421, 114), (469, 127)$ *generates the same points as the simpler-looking* $(1, 0), (0, 1)$ i. e. *all integral points.*

Let us give some general facts about lattices. With a basis $(\vec{b}_i)_{i=1}^n$ we associate a matrix B which has $\vec{b}_i$ as its i'th row. Since we assume that $(\vec{b}_i)_{i=1}^n$ are linearly independent over the rational numbers the determinant of B is nonzero. For any other basis we get a similar matrix B' and they span the same lattice if for some integer matrices M and M' we have that $B = MB'$ and $B' = M'B$. Clearly in such a case $M' = M^{-1}$ and since we are considering integral matrices the determinants of M and M' are both ± 1. This implies that $|det(B)| = |det(B')|$ and thus this number is independent of the chosen basis. This number is simply called the determinant of the lattice and is thus defined by

$$\det(L) = |\det \begin{pmatrix} \vec{b}_1 \\ \vec{b}_2 \\ \vdots \\ \vec{b}_n \end{pmatrix} |$$

for any basis $(\vec{b}_i))_{i=1}^n$. Since a determinant is equal to the volume of the parallelepiped spanned by the row vectors, the determinant of a lattice is in fact equal to the volume of the fundamental domain of the lattice. It is hence also the reciprocal of the "density" of L, where density is the number of points per volume in large spheres. Also, from this characterization it follows that this number does not depend on the particular choice of basis. In Example 9.3 we have that the determinant is

$$\begin{vmatrix} 1 & 2 \\ -1 & 3 \end{vmatrix} = 5$$

and thus we expect it to contain a fifth of all points, which is clearly seen from the other definition of the same lattice (*i. e.* the definition that the inner product with (3,1) should be divisible by 5). On the other hand, in Example 9.4

$$\begin{vmatrix} 421 & 114 \\ 469 & 127 \end{vmatrix} = 1.$$

Since the only integer lattice of density 1 is given by all integral points we can conclude, without doing any further calculations, that these vectors span all integral points.

We now turn to the general problem of finding a good basis for a lattice. The only change of basis that we use is to replace some $\vec{b}_i$ by $\vec{b}_i - \sum_{j \neq i} a_j \vec{b}_j$ for some integers a_j. These changes preserve the lattice. Let us first look at the case of dimension 2. The experience from Example 9.4 makes the following a natural suggestion:

```
GAUSS(b1, b2)
repeat
    arrange that b1 is the shorter vector
    find k ∈ Z such that b2 − k b1 is of minimal
    length
    b2 ← b2 − k b1
until k = 0
return b1, b2
```

Indeed, the output from Gauss is such that $\vec{b}_1$ is the shortest vector in the lattice and $\vec{b}_2$ is roughly orthogonal to $\vec{b}_1$. A simple geometric argument reveals that the angle between the output vectors is in the range $90° \pm 30°$. We omit the details. The reason for calling the procedure Gauss is that in fact it was described by Gauss in the last century.

It is far from clear how to extend the basis reduction to higher dimensions. Before we try let us just give an small example.

Example 9.5. *Now, let $n = 3, p = 5, k = 3$ and $h \equiv x + 57 \bmod p^k$ be parameters used to construct the lattice used in the polynomial factorization chapter. Then,*

$$\begin{aligned} h &\sim (57, 1, 0) = \vec{b}_1, \\ xh &\sim (0, 57, 1) = \vec{b}_2, \\ p^k e_1 &\sim (125, 0, 0) = \vec{b}_3 \end{aligned}$$

happen to form a basis for the needed lattice. After the operations

$$\begin{aligned} \vec{b}'_3 &= \vec{b}_3 - 2\vec{b}_1 = (11, -2, 0), \\ \vec{b}'_1 &= \vec{b}_1 - 5\vec{b}'_3 = (2, 11, 0), \\ \vec{b}'_2 &= \vec{b}_2 - 5\vec{b}'_1 = (-10, 2, 1) \end{aligned}$$

we have $\vec{b}'_2 + \vec{b}'_3 = (1, 0, 1) \sim x^2 + 1$, which certainly is a multiple of $h \bmod p^k$ whose coefficients are conveniently small. Note that $(x + 57)(x - 57) \equiv x^2 + 1 \bmod 125$.

Note the rather ad hoc nature of the reduction steps in the last example. We are in need of a more general technique.

9.2 General Basis Reduction

To generalize the basis reduction algorithm to any dimension was a great achievement, but the basic building blocks are simple. The basic idea is more or less the expected. We simply use recursion and all we need is to reduce the dimension of the problem. The latter is achieved by taking orthogonal projections. We have to be slightly careful, however, to avoid an exponential running time.

Assume we have a basis $\vec{b}_1, \vec{b}_2, \ldots, \vec{b}_n$ to reduce. Define

$$\begin{aligned}
\vec{b}_1^* &= \vec{b}_1, \\
\vec{b}_i^* &= \vec{b}_i \text{ projected orthogonally to } \vec{b}_1, \ldots, \vec{b}_{i-1}, 2 \le i \le n, \\
\vec{e}_i^* &= \text{ unit vector in the direction of } \vec{b}_i^*, \\
\beta_i &= |\vec{b}_i^*|.
\end{aligned}$$

What we have done is called a *Gram-Schmidt orthogonalization* and in the orthogonal basis e^* the basis is as follows:

$$\begin{aligned}
\vec{b}_1 &= (\beta_1, 0, \ldots, 0), \\
\vec{b}_2 &= (\mu_{12}, \beta_2, 0, \ldots, 0), \\
&\vdots \\
\vec{b}_n &= (\mu_{1n}, \mu_{2n}, \ldots, \mu_{(n-1)n}, \beta_n).
\end{aligned}$$

The following lemma shows one importance of the numbers β_i.

Lemma 9.6. *For any vector $\vec{v} \neq 0^n$ such that $\vec{v} \in L$, we have $\|\vec{v}\| \ge \min_{i=1}^n \beta_i$.*

Proof. Assume that $\vec{v} = \sum_{i=1}^n a_i \vec{b}_i$ and then i_0 is the largest coordinate such that $a_i \neq 0$. The then i_0'th coordinate of $\vec{v}$ in the basis $\vec{e}_i^*$ is $a_{i_0}\beta_{i_0}$ and hence

$$\|\vec{v}\| \ge |a_{i_0}\beta_{i_0}| \ge \min_{i=1}^{n} \beta_i.$$

□

We are now ready for the algorithm L^3, so called because it first appeared in a paper by Lenstra, Lenstra and Lovász [34]. This paper gave the first fully proved polynomial time algorithm for factorization of integer polynomials. The subroutine for finding a short vector in a lattice is due to Lovász alone and thus the name L^3 is slightly misleading but also firmly established. In view of Lemma 9.6 and the fact that we want $\vec{b}_1$ to be the short vector we output, the key to the algorithm is to make sure that no β_i is much smaller than β_1 which is the length of $\vec{b}_1$. It turns out that if β_{i+1}^2 is smaller than $\frac{1}{2}\beta_i^2$ we can interchange $\vec{b}_i$ and $\vec{b}_{i+1}$ to make β_i smaller. In this way we push the small values to the smaller indices. The algorithm is given in detail below.

Input: a basis and its orthogonalization
Output: a reduced basis
(1) arrange that $|\mu_{ij}| \le \frac{1}{2}\beta_i$, **for** $1 \le i \le n-1, 2 \le j \le n$
(2) **if** $\exists i : \beta_{i+1}^2 \le \frac{1}{2}\beta_i^2$
(3) exchange $\vec{b}_i$ and $\vec{b}_{i+1}$
(4) **goto** 1

Before we analyze the algorithm let us just describe in detail how to do the first step. The key is the order in which to take care of the conditions, we first make sure that $\mu_{(n-1)n}$ is small by subtracting a suitable integer multiple of $\vec{b}_{n-1}$ from $\vec{b}_n$. We then take care of $\mu_{(n-2)(n-1)}$ and $\mu_{(n-2)n}$ by subtracting a suitable integer multiple of $\vec{b}_{n-2}$ from $\vec{b}_{n-1}$ and $\vec{b}_n$ respectively. Note that this does not change $\mu_{(n-1)n}$! We then take care of $\mu_{(n-3)j}$ etc. The process thus resembles Gaussian elimination.

Our analysis has to answer two questions:

1. How good is the result, *i. e.* how much longer is the vector $\vec{b}_1$ compared to the shortest vector on termination?

2. How many iterations are needed in the worst case and how long time does an iteration take?

The first question is essentially answered by

Lemma 9.7. *Upon termination the length of the vector $\vec{b}_1$ is at most $2^{(n-1)/2}$ times the length of the shortest vector in L.*

Proof. The termination condition gives:

$$\beta_1^2 \leq 2\beta_2^2 \leq \ldots \leq 2^{n-1}\beta_n^2,$$

and by Lemma 9.6 we have

$$|\vec{b}_1| = \beta_1 \leq 2^{\frac{n-1}{2}} \min_{i=1}^{n} \beta_i \leq 2^{\frac{n-1}{2}} |\text{the shortest vector in L}|.$$

□

To answer the second question let us first analyze the time needed for one iteration. The most time-consuming step is to make sure that the μ_{ij} are small which is similar to Gaussian elimination, and takes time $O(n^3)$. Next we bound the number of iterations. Let us consider an interchange and let $\vec{b}'_i$ denote the new value of β_i (and similarly for other quantities). Suppose we exchange $\vec{b}_2$ and $\vec{b}_3$. After the exchange, we have $\beta'_2 = |\vec{b}_2^{*\prime}|$ = the length of the part of $\vec{b}'_2$ that is orthogonal to $\vec{b}_1$, but $\vec{b}'_2 = \vec{b}_3$, so

$$\beta'_2 = \sqrt{\beta_3^2 + \mu_{23}^2} \leq \sqrt{\frac{1}{2}\beta_2^2 + \frac{1}{4}\beta_2^2} = \sqrt{\frac{3}{4}}\beta_2.$$

Now β_4 is the part of $\vec{b}_4$ which is orthogonal to $\vec{b}_1$, $\vec{b}_2$ and $\vec{b}_3$. Since this set of vectors does not change when we interchange b_2 and b_3, β_4 remains unchanged and if fact for $j \geq 4$, we have $\beta'_j = \beta_j$ and since $\det(L) = \prod_{i=1}^{n} \beta_i = \prod_{i=1}^{n} \beta'_i$, it follows that

$$\beta'_3 = \frac{\beta_2\beta_3}{\beta'_2} \geq \sqrt{\frac{4}{3}}\beta_3.$$

So we conclude that an exchange at i only affects β_i and β_{i+1}, of which the first decreases and the latter increases. Furthermore, we have lower-bounded their rate of change. Consider how an exchange affects the product $B = \prod_{j=1}^{n} \beta_j^{n-j}$. The only interesting part is

$$\beta_i^{n-i}\beta_{i+1}^{n-i-1} = \beta_i \underbrace{(\beta_i\beta_{i+1})^{n-i-1}}_{\text{unchanged}},$$

so B decreases by the same factor as does the particular β_i in each iteration.

If the longest input vector has length D, then initially, $\beta_j \leq D$ which implies that to begin with

$$B \leq \prod_{j=1}^{n} D^{n-j} = D^{\frac{n(n-1)}{2}},$$

and in the end $B \geq 1$, because

$$\begin{aligned}
\beta_1 &\geq 1 \text{ (length of } \vec{b}_1) \\
\beta_1\beta_2 &\geq 1 \text{ (det} \left(\begin{pmatrix} \vec{b}_1 \\ \vec{b}_2 \end{pmatrix} \begin{pmatrix} \vec{b}_1^T & \vec{b}_2^T \end{pmatrix} \right) = \beta_1^2\beta_2^2\text{, an integer)} \\
&\vdots \\
\beta_1\beta_2\cdots\beta_j &\geq 1 \\
&\vdots \\
\beta_1\beta_2\cdots\beta_n = \det(L) &\geq 1,
\end{aligned}$$

and B is the product of these numbers.

Thus we see that the number of iterations is in $O(n^2 \log D)$, and that the total running time of the algorithm is in $O(n^5 \log D)$ operations on numbers. To complete the analysis we would need to analyze the accuracy needed during the computations. This is rather tedious and we omit this analysis. In practice, however, reports show that one has to be rather accurate and use at least double precision in larger dimensions. This answers the second question and concludes our analysis of L^3. We end by stating a formal theorem.

Theorem 9.8. *Given a basis of a lattice in n dimensions where each of the basis vectors is of length at most D, then the algorithm L^3 finds a vector in L which is at most $2^{(n-1)/2}$ times longer than the shortest vector. The algorithm runs in time $O(n^5 \log D)$ operations on numbers.*

Chapter 10

Multiplication of Large Integers

The computational problem of multiplying extremely large integers might not seem of immediate practical use since the standard method is sufficient while the numbers only have a few thousand digits. However, extremely large numbers do come up in some applications and in those cases an efficient multiplication algorithm is of vital importance. There is also another motivation of a more philosophical nature. Multiplication is a basic operation and it is fundamental to our understanding of efficient computation to investigate how to multiply in the most efficient matter. With this in mind it is also curious to note that the fastest algorithm given in these notes (taken from [47]), although it is the fastest algorithms known, is not known to be optimal. Thus there might still be more efficient algorithms for multiplication to be discovered!

The standard way to multiply large numbers that we all learned in grade school uses $O(n^2)$ steps to multiply two n-bit numbers. The first algorithm improving this bound was given by Karatsuba [29] and since it is rather simple we present this algorithm first.

10.1 Karatsuba's algorithm

We want to multiply the two n bit numbers x and y. Suppose that n is a power of 2 and rewrite x and y:

$$x = a + 2^{n/2}b, \; y = c + 2^{n/2}d,$$

where $a, b, c, d < 2^{n/2}$. What we want to calculate is the product

$$xy = (a + 2^{n/2}b)(c + 2^{n/2}d) = ac + 2^{n/2}(ad + bc) + 2^n bd.$$

Noticing that $ad + bc = (a+b)(c+d) - ac - bd$, it suffices to compute three $n/2$[1] bit products: ac, bd, and $(a+b)(c+d)$, which can be calculated by recursion. We also need a few additions and shift operations in each recursion level. Since

[1] The numbers $(a+b)$ and $(c+d)$ might have $n/2+1$ bits and to be exact this extra bit should be handled separately. It does not change the analysis however and we ignore this little complication.

the latter can be done in time $O(n)$ if we denote the number of bit operations needed to multiply two n bit numbers by $M(n)$, we get the following recurrence relation.

$$M(n) = 3M(n/2) + O(n).$$

A recurrence relation of the form $M(n) = aM(n/b) + O(n^c)$ has the solution $M(n) = O(n^{\log_b a})$ provided $c < \log_b a$ we get in this case that $M(n) = O(n^{\log_2 3})$ and since $\log_2 3 \approx 1.58$ this is considerably faster than the naive algorithm. We state this as a theorem before we proceed to develop tools for even faster algorithms.

Theorem 10.1. *Two n-bit numbers can be multiplied using $O(n^{\log_2 3})$ bit operations.*

10.2 The Discrete Fourier Transform

The discrete Fourier transform is a basic building block in many algorithms. We use it to get a fast algorithm for multiplying large numbers, but the main application in real life is in signal processing.

10.2.1 Definition

The discrete Fourier transform $\hat{a}_0, \hat{a}_1, \ldots, \hat{a}_{n-1}$ of the sequence $a_0, a_1, \ldots, a_{n-1}$ is defined by

$$\hat{a}_j = \sum_{i=0}^{n-1} \omega^{ij} a_i,$$

where ω is an n'th root of unity, *i. e.* , $\omega^n = 1$ and $\omega^k \neq 1, k < n$. Most of the time ω will be the complex number $e^{2\pi i/n}$, but we will also consider other domains. The inverse of the transform is

$$a_k = \frac{1}{n} \sum_{j=0}^{n-1} \omega^{-jk} \hat{a}_j.$$

We leave it to the reader to verify that this is in fact a correct inversion.

One of the reasons to use the Fourier transform is that convolution of two sequences is transformed to pairwise multiplication in the transform domain:

$$c = a * b \sim \hat{c}_k = \hat{a}_k \hat{b}_k,$$

where convolution is defined by

$$a * b = \sum_{i+j \equiv k \bmod n} a_i b_j.$$

We leave the proof of also this fact as an exercise.

10.3 The Fast Fourier Transform

To calculate the Fourier transform in an efficient way, we use a method known as the fast Fourier transform (FFT). To see how the FFT works, suppose that n is a power of 2 and split the definition sum into terms with even index and those with odd index.

$$\hat{a}_j = \sum_{i=0}^{n-1} \omega^{ij} a_i = \sum_{k=0}^{n/2-1} \omega^{2kj} a_{2k} + \sum_{k=0}^{n/2-1} \omega^{(2k+1)j} a_{2k+1}.$$

If we put $\sigma = \omega^2$ then σ is an $n/2$'th root of unity and we can use if for transforms of size $n/2$. We use it to transform the two sequences given by the odd and even terms of our original sequence. Let $(\hat{a}^e_j)_{j=0}^{n/2-1}$ be the transform of the even elements, *i. e.*

$$\hat{a}^e_j = \sum_{k=0}^{n/2-1} \sigma^{jk} a_{2k}$$

and similarly let

$$\hat{a}^o_j = \sum_{k=0}^{n/2-1} \sigma^{jk} a_{2k+1}$$

be the transform of the odd elements. Then by straightforward identification of terms we have

$$\hat{a}_j = \hat{a}^e_j + \omega^j \hat{a}^o_j,\ j = 0, 1, \ldots, n/2 - 1, \tag{10.1}$$

and

$$\hat{a}_j = \hat{a}^e_{j-n/2} + \omega^j \hat{a}^o_{j-n/2},\ j = n/2, n/2 + 1, \ldots, n - 1. \tag{10.2}$$

Thus we can calculate the Fourier transform with the help of a divide and conquer strategy. By recursion we compute the transforms of the odd and even terms and then we use (10.1) and (10.2) to complete the calculation. Since the latter costs $O(n)$ arithmetic operations, if we let $T(n)$ be the number of operations needed for a transform of size n we get

$$T(n) = cn + 2T(n/2)$$

and this solves to $T(n) = O(n \log n)$. Let us state this as a theorem

Theorem 10.2. *When n is a power of 2, the DFT can be computed by the FFT in $O(n \log n)$ arithmetic operations.*

10.4 Fast Multiplication

The basic idea to make a faster algorithm is to use the Fourier transform: the main work in multiplying resembles convolution, so by using the Fourier transform, we can multiply the numbers bitwise in the transform domain.

10.4.1 A First Attempt

Our first attempt to make an algorithm for multiplication is the following:

> **Input:** Two numbers a and b represented by the binary sequences $a_0, a_1, \ldots, a_{n-1}$ and $b_0, b_1, \ldots, b_{n-1}$.
> **Output:** The binary sequence $c_0, c_1, \ldots, c_{n-1}$ representing the product $c = ab$.
> (1) Calculate the transforms $\hat{a}, \hat{b}$.
> (2) $\hat{c}_k \leftarrow \hat{a}_k \hat{b}_k$
> (3) Compute c by doing the inverse transformation.

Step 1 uses $O(n \log n)$ exact complex multiplications. Step 2 can be calculated in $O(n)$ operations and step 3 takes the same time as step 1.

A minor problem is that

$$c_0 = a_0 b_0 + a_1 b_{n-1} + a_2 b_{n-2} + \cdots$$

is not really the least significant bit of the product. To get the correct answer we have to use $2n$ bits, and to extend the input vectors to double length by adding coordinates which are all set to zero. In other words we change n to $2n$ by setting $a_{n+i} = b_{n+i} = 0$ for $0 \leq i \leq n-1$. Another minor problem is then that the c_i are not the bits of the products, but rather numbers from 0 to n. Thus we have to add these n numbers, each consisting of $\log n$ bits but this can easily be done in time $O(n \log n)$ and thus this is not a problem either.

What we really want is to derive a method for multiplication that is fast measured in the number of bit operations needed. So, obviously, using exact complex arithmetics is not desirable. There seem to be two possible solutions:

1. Use the method above, but try to limit the need of accuracy in the complex multiplications.

2. Try to use the Fourier transform over some other domain than the complex numbers, *e. g.* , over $\mathbb{Z}_N$ for some N.

10.4.2 Use complex numbers

If we want to use the complex Fourier transform as proposed above, the number of bits of accuracy needed turns out to be $O(\log n)$. Let us sketch why this is sufficient. Suppose b bits of accuracy are used in representing the numbers ω^j and in the calculations. If two numbers, each known with j bits of accuracy are added or multiplied the result is known with about $j-1$ bits of accuracy. This implies that at the i'th level of the recursion, we have $b - O(i)$ bits of accuracy in the numbers. Since the recursion is $\log n$ levels deep we have $b - O(\log n)$ bits of accuracy in then answer.

Now if a_i and b_i are used as individual bits we know that the answers c_j are integers which are at most n and thus $2 + \log n$ bits of accuracy in the answer is sufficient to obtain the exact answer. We conclude that $b = d \log n$ for a suitable constant d is sufficient and let us analyze the resulting algorithm.

If $M(n)$ is the number of bit operations needed to multiply two n bit numbers, we get

$$\begin{aligned} M(n) &= cn\log n M(O(\log n)) \\ &= cn\log n\, c'\log n \log\log n M(O(\log\log n)) \\ &= O(n(\log n)^2 \log\log n) M(O(\log\log n)) \\ &\in O(n(\log n)^{2+\epsilon}). \end{aligned}$$

It turns out to be better to have every a_i and b_i consist of $\log n$ bits each. We then take convolution of vectors of length $n/\log n$, containing integers $0, 1, \ldots, n-1$; and by the above reasoning $O(\log n)$ bits of accuracy still suffices and we get.

$$\begin{aligned} M(n) &= c(n/\log n)\log(n/\log n) M(O(logn)) \\ &\approx cnM(O(\log n)) \\ &= O(n\log n) M(O(\log\log n)) \\ &\in O(n\log n(\log\log n)^{1+\epsilon}). \end{aligned}$$

This is the best known running time upto the existence of ϵ. We show below how to eliminate this ϵ. In practice, however, we recommend the algorithm with complex numbers since in most applications the algorithm can be performed with single or at most double precision and then there is no reason to worry about recursion, and real life computers are slowed by the need to use recursion.

Example 10.3. *We want to FFT multiply the integers $A = 1634$ and $B = 9827$.*
$a_0 \ldots a_7 = (4, 3, 6, 1, 0, 0, 0, 0)$
$b_0 \ldots b_7 = (7, 2, 8, 9, 0, 0, 0, 0)$

We want to do a 8-dimensional FFT using the 8'th root of unity $\omega = e^{2\pi i/8} = \sqrt{1/2} + i\sqrt{1/2}$. We want to perform calculations to a fixed precision and we use two decimal digits of accuracy and use the value $.71 + .71i$ for ω, and similar values for larger powers of ω. We note, however, that $\omega^2 = i$ and $\omega^4 = -1$ and hence there is no need to use approximate values for these numbers. If we unravel the FFT we see that the final result is in fact explicitly given by

$$\hat{a}_0 = \hat{a}_0^e + \omega^0\hat{a}_0^o \qquad \hat{a}_1 = \hat{a}_1^e + \omega^1\hat{a}_1^o \qquad \hat{a}_2 = \hat{a}_2^e + \omega^2\hat{a}_2^o \qquad \hat{a}_3 = \hat{a}_3^e + \omega^3\hat{a}_3^o$$
$$\hat{a}_4 = \hat{a}_0^e + \omega^4\hat{a}_0^o \qquad \hat{a}_5 = \hat{a}_1^e + \omega^5\hat{a}_1^o \qquad \hat{a}_6 = \hat{a}_2^e + \omega^6\hat{a}_2^o \qquad \hat{a}_7 = \hat{a}_3^e + \omega^7\hat{a}_3^o$$

where $(\hat{a}_i^e)_{i=0}^3$ is the transform of the even part $(4, 6, 0, 0)$ and $(\hat{a}_i^o)_{i=1}^3$ is the transform of the odd part (3, 1, 0, 0). We calculate these two transforms by recursion using $\sigma = \omega^2 = i$ as the appropriate root of unity. We start with the even part $(4, 6, 0, 0)$ and again we have after expanding

$$\hat{a}_0^e = \hat{a}_0^{ee} + \sigma^0\hat{a}_0^{eo} \qquad \hat{a}_1^e = \hat{a}_1^{ee} + \sigma^1\hat{a}_1^{eo} \qquad \hat{a}_2^e = \hat{a}_0^{ee} + \sigma^2\hat{a}_0^{eo} \qquad \hat{a}_3^e = \hat{a}_1^{ee} + \sigma^3\hat{a}_0^{eo},$$

where $(\hat{a}_0^{ee}, \hat{a}_1^{ee})$ is the transform of the even even part i.e. $(4, 0)$ and $(\hat{a}_0^{eo}, \hat{a}_1^{eo})$ is the transform of $(6, 0)$. Using the definition of a transform of size two with the root of unity $\gamma = \omega^4 = -1$ we have

$$\begin{aligned} \hat{a}_0^{ee} &= \gamma^{0\cdot 0}\cdot 4 + \gamma^{1\cdot 0}\cdot 0 = 4 \\ \hat{a}_1^{ee} &= \gamma^{0\cdot 1}\cdot 4 + \gamma^{1\cdot 1}\cdot 0 = 4 \\ \hat{a}_0^{eo} &= \gamma^{0\cdot 0}\cdot 6 + \gamma^{1\cdot 0}\cdot 0 = 6 \\ \hat{a}_1^{eo} &= \gamma^{0\cdot 1}\cdot 6 + \gamma^{1\cdot 1}\cdot 0 = 6. \end{aligned}$$

Substituting this back into the previous expressions we get

$$\begin{aligned}
\hat{a}^e_0 &= 4 + \sigma^0 \cdot 6 = 10 \\
\hat{a}^e_1 &= 4 + \sigma^1 \cdot 6 = 4 + 6i \\
\hat{a}^e_2 &= 4 + \sigma^2 \cdot 6 = 4 - 6 = -2 \\
\hat{a}^e_3 &= 4 + \sigma^3 \cdot 6 = 4 - 6i.
\end{aligned}$$

Let us now turn to the odd part, $(3, 1, 0, 0)$*. We have, with similar notation, as before*

$$\hat{a}^o_0 = \hat{a}^{oe}_0 + \sigma^0 \hat{a}^{oo}_0 \qquad \hat{a}^o_1 = \hat{a}^{oe}_1 + \sigma^1 \hat{a}^{oo}_1 \qquad \hat{a}^o_2 = \hat{a}^{oe}_0 + \sigma^2 \hat{a}^{oo}_0 \qquad \hat{a}^o_3 = \hat{a}^{oe}_1 + \sigma^3 \hat{a}^{oo}_0,$$

where $(\hat{a}^{oe}_0, \hat{a}^{oe}_1)$ *is the transform of the even even part i.e.* $(3, 0)$ *and* $(\hat{a}^{oo}_0, \hat{a}^{oo}_1)$ *is the transform of* $(1, 0)$*. By definition*

$$(\hat{a}^{oe}_0, \hat{a}^{oe}_1) = (3, 3) \qquad (\hat{a}^{oo}_0, \hat{a}^{oo}_1) = (1, 1)$$

giving

$$(\hat{a}^o_0, \hat{a}^o_1, \hat{a}^o_2, \hat{a}^o_3) = (4, 3 + i, 2, 3 - i).$$

Substituting this into the first set of equations and using the approximate value for the powers of ω *gives*

$$\begin{aligned}
\hat{a}_0 &= 10 + 4 = 14 \\
\hat{a}_1 &= 4 + 6i + \omega \cdot (3 + i) = 5.42 + 8.84i \\
\hat{a}_2 &= -2 + \omega^2 \cdot 2 = -2 + 2i \\
\hat{a}_3 &= 4 - 6i + \omega^3 \cdot (3 - i) = 2.58 - 3.16i \\
\hat{a}_4 &= 10 + \omega^4 4 = 6 \\
\hat{a}_5 &= 4 + 6i + \omega^5 \cdot (3 + i) = 2.58 + 3.16i \\
\hat{a}_6 &= -2 + \omega^6 \cdot 2 = -2 - 2i \\
\hat{a}_7 &= 4 - 6i + \omega^7 \cdot (3 - i) = 5.42 - 8.84i
\end{aligned}$$

Note that the pairs $(\hat{a}_1, \hat{a}_7)$*,* $(\hat{a}_2, \hat{a}_6)$*,* $(\hat{a}_3, \hat{a}_5)$ *are the complex conjugates of each other. This is no surprise since this property of getting conjugate pairs is true for any real initial vector* a*. We now calculate the transformation of* b *in the same way yielding the result.*

$$\hat{b} = (26, 2.03 + 15.81i, -1 - 7i, 11.97 - 0.19i, 4, 11.97 + 0.19i, -1 + 7i, 2.03 - 15.81i)$$

We multiply the two transforms componentwise $(\hat{c}_i = \hat{a}_i \cdot \hat{b}_i)$ *to get*

$$\hat{c} = (364, -128.76 + 103.64i, 16 + 12i, 30.28 - 38.32i, 24, 30.28 + 38.32i, 16 - 12i, -128.76 - 103.64i).$$

Now do the inverse FFT on $\hat{c}$ *which is similar to the forward transform except that we replace* ω *by* $\omega^{-1} = \sqrt{1/2} - i\sqrt{1/2}$*. We also divide the final result by 8 obtaining*

$$c = (27.88, 28.86, 79.99, 79.32, 77.12, 62.13, 9.01, -0.32).$$

We know that the components of c *are integers and hence we round them to*

$$c = (28, 29, 80, 79, 77, 62, 9, 0)$$

which in fact is the correct answer. We note in all honesty that the distance from the final components to the nearest integers were a bit too large to be really sure that the answers are correct. It would have been wise to redo the calculation with 3 digits of accuracy which yields

$$c = (28.004, 29.005, 80.000, 78.988, 76.995, 61.995, 8.999, 0.011)$$

which certainly looks much nicer. In any case we know that the correct product of the two given numbers is

$$\sum_{i=0}^{7} c_i 10^i$$

but our "digits" c_i *are larger than 10 and hence we move the overflow to higher index* c_i *one by one giving the process*

$$\begin{aligned}
c &= (8, 31, 80, 79, 77, 62, 9, 0) \\
c &= (8, 1, 83, 79, 77, 62, 9, 0) \\
c &= (8, 1, 3, 87, 77, 62, 9, 0) \\
c &= (8, 1, 3, 7, 85, 62, 9, 0) \\
c &= (8, 1, 3, 7, 5, 70, 9, 0) \\
c &= (8, 1, 3, 7, 5, 0, 16, 0) \\
c &= (8, 1, 3, 7, 5, 0, 6, 1),
\end{aligned}$$

yielding the answer 16057318. If we keep track of constants we see that the cost to multiply two n *digit numbers is something like* $6n \log n$ *complex multiplications. The traditional method requires* n^2 *multiplications of individual digits and a few additions. Since* $6n \log n$ *is larger than* n^2 *for small* n *the benefit is not visible in such a small example as* $n = 8$*. However, as* n *gets large the situation changes drastically.*

10.4.3 Making calculations in $\mathbb{Z}_N$

This time we do the transform in the ring $\mathbb{Z}_N$ instead. We require that $n = 2^k$. First let $B = 2^{\lfloor k/2 \rfloor}$ and $l = n/B$. We let the input vectors a and b be of length B containing integers $0, 1, \ldots, 2^l - 1$. Calculations are to be performed modulo $N = 2^{2l} + 1$. This gives the algorithm for integer multiplication with the best known asymptotic running time. It is called the Schönhage-Strassen algorithm after its inventors.

The number $2^{4l/B}$ is a Bth unit root since $(2^{4l/B})^B = 2^{4l} = (2^{2l})^2 = (-1)^2 = 1 \bmod 2^{2l} + 1$. Thus since this is the only property we need of ω, $\omega = 2^{4l/B}$ is a valid choice.

We represent a number modulo $2^{2l} + 1$ by $2l$ bits. We use that $x2^{2l} + y \equiv y - x \bmod 2^{2l} + 1$ so that multiplication by powers of the form $2^{4l/B}$ and addition can be done in $O(l)$ bit operations.

Let us analyze our proposed multiplication algorithm given in Section 10.4.1. We find that step 1 requires $O(B \log B)$ multiplications, each of which can be performed in $O(l)$ bit operations. The multiplication of the transforms in step 2 requires $BM(2l)$ operations where $M(n)$ denotes the number of operations needed to multiply numbers of length n. We get

$$\begin{aligned} M(n) &= O(lB\log B) + BM(2l) \\ &= O(n\log n) + BM(2l) \\ &= cn\log n + BM(2l), \end{aligned}$$

where we set $M(n) = M'(n)n$:

$$\begin{aligned} nM'(n) &= cn\log n + B2lM'(2l) \\ M'(n) &= c\log n + 2M'(2l) \\ &\leq c\log n + 2M(4\sqrt{n}) \\ &\leq c\log n + 2c\log 4\sqrt{n} + 4M'(4\sqrt{4\sqrt{n}}) \\ &\leq c\log n + c\log n + 4c\log(4\sqrt{4\sqrt{n}}) + 8M'(dn^{1/8}). \end{aligned}$$

We reach $M'(1)$ after $\log\log n$ steps; and thus the total cost is be $O(n\log n\log\log n)$.

Unfortunately, the method described above needs some minor adjustments. A few things to worry about are:

1. If the components of the result are larger than 2^{2l} the algorithm gives an erroneous result; c_k might be as large as $B2^{2l}$ and we only know c_k modulo $2^{2l}+1$.
2. What happens if some number becomes exactly 2^{2l}? $2l$ bits are not enough to represent such a number.
3. How do we convert the output vector to a number? Is knowledge of c_k enough to calculate the product?
4. We want to determine a product. In the recursion we calculate the product modulo $2^{2l}+1$.

Problem 2 is a simple special case; we do not draw any further attention to it. Problem 4 is solved by defining our basic computational problem to compute a product of two integers modulo 2^n+1 for a given n. When computing the product of two n-bit integers there is no harm to compute the product mod $2^{2n}+1$ and thus we can simply double n on the first level of recursion.

When we multiply a and b represented by the vectors $a = (a_0, a_1, \ldots, a_{B-1})$ and $b = (b_0, b_1, \ldots, b_{B-1})$:

$$a = \sum_{i=0}^{B-1} 2^{li}a_i$$

and

$$b = \sum_{i=0}^{B-1} 2^{li}b_i,$$

we want to get the product

$$ab = \sum_{k=0}^{2B-2}\left(2^{kl}\sum_{i+j\equiv k} a_ib_j\right) \equiv \sum_{k=0}^{B-1} 2^{kl}c'_k \bmod 2^n+1$$

where

$$c'_k = \sum_{i+j=k} a_i b_j - \sum_{i+j=k+B} a_i b_j.$$

To calculate c'_k, use the vectors

$$\begin{array}{rcl} a'' & = & (a_0 \quad \Psi a_1 \quad \Psi^2 a_2 \quad \dots \quad \Psi^{B-1} a_{B-1}), \\ b'' & = & (b_0 \quad \Psi b_1 \quad \Psi^2 b_2 \quad \dots \quad \Psi^{B-1} b_{B-1}), \end{array}$$

where $\Psi^B = -1$ ($\Psi = 2^{2l/B}$, 4 or 16). If we use the Fourier transform to compute the convolution of a'' and b'' ($\hat{c}''_k = \hat{b}''_k \hat{a}''_k$), we get

$$c'' \quad = \quad (c'_0 \quad \Psi c'_1 \quad \Psi^2 c'_2 \quad \dots \quad \Psi^{B-1} c'_{B-1}).$$

It is now easy to calculate the sum

$$\sum_{k=0}^{B-1} 2^{kl} c'_k$$

in $O(n)$ operations. This solves problem 3.

For problem 1 we know $c'_k \bmod 2^{2l}+1$. If we calculate $c'_k \bmod B$ we also know $c'_k \bmod B(2^{2l}+1)$:

$$c'_k \equiv \begin{cases} x \bmod 2^{2l}+1 \\ y \bmod B \end{cases}$$

$$\Rightarrow c'_k = (y-x)(2^{2l}+1) + x.$$

We can calculate $c'_k \bmod B$ as

$$c'_k = \sum_{i+j=k} a_i b_j - \sum_{i+j=k+B} a_i b_j \bmod B.$$

We have to deal with $\log B$ bit numbers, to compare with B bit numbers that we have used until now, so this should be fairly simple. Let $A_i = a_i \bmod B$ and $B_i = b_i \bmod B$, and let

$$\alpha = \sum_{i=0}^{B-1} B^{3i} A_i,$$

$$\beta = \sum_{i=0}^{B-1} B^{3i} B_i.$$

Now calculate the product $\alpha\beta$, which is the product of two $O(B \log B)$ bit numbers, for example, by using Karatsuba's algorithm in time $O((B \log B)^{1.58}) \subset o(n)$ operations:

$$\alpha\beta = \sum_{k=0}^{2B-2} B^{3k} \underbrace{\sum_{i+j=k} A_i B_j}_{\leq B^3} = \sum_{k=0}^{2B-2} B^{3k} \gamma_k.$$

We can now calculate $c'_k \bmod B$ as $\gamma_k - \gamma_{k+B} \bmod B$. This takes care of all our problems.

10.5 Division of large integers

Let x and y be two numbers each with at most n bits. We let division be the problem of computing an approximation of x/y with n digits accuracy, i.e. of finding a number q such that $|x/y - q| \leq 2^{-n}q$. Intuitively, most people think of division as the most difficult arithmetical operation, but from a complexity theory point of view it turns out not to be much more complicated than multiplication.

Theorem 10.4. *Let $M(n)$ be a function such that for any n, $2M(n) \leq M(2n) \leq CM(n)$ for some constant C and such that multiplication of two n bit integers can be done in time $M(n)$. Then, we can also solve division in time $O(M(n))$.*

The technical assumption $2M(n) \leq M(2n) \leq CM(n)$ is very mild and is true for any nice function that grows at least linearly and at most quadratically.

Let us now describe the algorithm. Firstly, note that it is sufficient to compute $1/y$ with n (or possibly $n+1$ but we do not care about such small details) bits of accuracy. A final multiplication by x of cost $M(n)$ would then give the full answer. To simplify notation we divide y by a power of 2 to get $1/2 \leq y \leq 1$ given by a binary expansion with at most n bits. Define $x_0 = 1$ and let

$$x_{i+1} = 2x_i - yx_i^2.$$

We have the following lemma:

Lemma 10.5. *We have $|x_i y - 1| \leq 2^{-2^i}$.*

Proof. We proceed by induction over i. The lemma is clearly true for $i = 0$. For general i we have

$$1 - x_{i+1}y = 1 - 2x_i y + y^2 x_i^2 = (1 - x_i y)^2,$$

which completes the induction step. □

The lemma implies that with $i = 1 + \log n$, x_i is an approximation to $1/y$ within the desired accuracy. Since each iteration can be done with two multiplications and one subtraction this would give an algorithm for inversion that runs in time $O(M(n)\log n)$. To remove this unwanted factor of $\log n$ we proceed as follows.

Since only the 2^i first bits of x_i are correct it seems wasteful to use n bits of accuracy in the calculations. Let $\tilde{y}_i$ be y with $5 + 2^i$ bits of accuracy, start with $\tilde{x}_0 = 1$, use the updating rule

$$x_{i+1} = 2\tilde{x}_i - y_i\tilde{x}_i^2$$

where $\tilde{x}_i$ is x_i rounded to $5 + 2^i$ bits. We claim that

$$|1 - \tilde{x}_i y_i| \leq 2^{-2^{i-1}-1/2}.$$

We again prove this by induction. It is true for $i = 0$. For the induction-step we note that

$$|\tilde{x}_{i+1}y_{i+1} - x_{i+1}y_i| \leq |\tilde{x}_{i+1}y_{i+1} - \tilde{x}_{i+1}y_i| + |\tilde{x}_{i+1}y_i - x_{i+1}y_i| \leq 2^{-(3+2^i)}$$

and that, as before,

$$1 - x_{i+1} y_i = (1 - \tilde{x}_i y_i)^2.$$

The two inequalities taken together yield

$$|1 - \tilde{x}_{i+1} y_{i+1}| \leq 2^{-3-2^i} + 2^{-2^i-1} \leq 2^{-2^i-1/2}$$

and the induction-step is complete.

The cost of the i'th iteration is $2M(5 + 2^i) + O(2^i)$ and thus we get total cost

$$\sum_{i=1}^{1+\log n} 2M(5 + 2^i) + O(2^i) \leq O(M(n))$$

where we have used the assumptions on the function M. Thus we can conclude that division is only marginally more expensive than multiplication and we have established Theorem 10.4.

The iteration formula might look like magic when first encountered but in fact this is not the case. Remember that Newton's method for solving a non-linear equation $f(x) = 0$ is to use iterations by the formula

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}.$$

If we apply this with $f(x) = y - \frac{1}{x}$, since $f'(x) = \frac{1}{x^2}$ we get the iterative formula we used. The quadratic convergence is then natural since this is true for Newton-iterations applied to smooth functions.

Chapter 11

Matrix multiplication

11.1 Introduction

Given two $n \times n$ matrices, A and B, we wish to examine how fast we can compute the product $AB = C$. The normal method takes time $O(n^3)$ since each entry in C is computed as

$$c_{i,k} = \sum_{j=1}^{n} a_{i,j} b_{j,k},$$

which means that each of the n^2 entries in C takes time $O(n)$ to compute.

The first algorithm to do better than this obvious bound was given by Strassen in 1969 [52]. The idea of this algorithm is to divide each $n \times n$ matrix into four parts of size $n/2 \times n/2$. If we view the submatrices of size $n/2 \times n/2$ as single elements, the problem is reduced to computing the product of two 2×2 matrices, and each subproblem can be solved recursively. The basic building block of the algorithm is therefore a computation of the product of two 2×2 matrices. The key to Strassen's algorithm is to compute such a product with 7 multiplications which is one multiplication fewer than the 8 multiplications of the standard algorithm. Let us describe Strassen's method. We want to compute

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}.$$

We first compute

$$\begin{aligned}
m_1 &= (a_{12} - a_{22})(b_{21} + b_{22}) \\
m_2 &= (a_{11} + a_{22})(b_{11} + b_{22}) \\
m_3 &= (a_{11} - a_{21})(b_{11} + b_{12}) \\
m_4 &= (a_{11} + a_{12})b_{22} \\
m_5 &= a_{11}(b_{12} - b_{22}) \\
m_6 &= a_{22}(b_{21} - b_{11}) \\
m_7 &= (a_{21} + a_{22})b_{11}.
\end{aligned}$$

The answer is now given by

$$\begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix},$$

where

$$\begin{aligned} c_{11} &= m_1 + m_2 - m_4 + m_6 \\ c_{12} &= m_4 + m_5 \\ c_{21} &= m_6 + m_7 \\ c_{22} &= m_2 - m_3 + m_5 - m_7. \end{aligned}$$

If $T(n)$ denotes the time this algorithm takes to multiply two $n \times n$ matrices, we hence get the recurrence relation $T(n) = 7T(n/2) + 18n^2$ which is solved by $T(n) \in O(n^{\log_2 7}) \in O(n^{2.81})$.

Today, the fastest known algorithm is by Coppersmith and Winograd [12] (1987), and runs in time $O(n^{2.376})$. In these notes we describe a algorithm with complexity $O(n^{2.67})$. Note, however, that these two algorithms are not practical because of the very large constants involved.

11.2 Bilinear problems

A problem on the following form is called *bilinear.* Inputs: x_i, $i \in I$, and y_j, $j \in J$ where I and J are index sets. Output: z_k, $k \in K$, where K is an index set and

$$z_k = \sum_{i,j} c_{i,j,k} x_i y_j.$$

Example 11.1. *The multiplication of two complex numbers x_1+ix_2 and y_1+iy_2 can be written as $(y_1 + ix_2)(y_1 + iy_2) = z_1 + iz_2$ where*

$$\begin{aligned} z_1 &= x_1y_1 - x_2y_2, \\ z_2 &= x_1y_2 + x_2y_1. \end{aligned}$$

Comparing this with the sum z_k above, we see that the coefficients $c_{i,j,1}$ can be written as the matrix

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix},$$

and $c_{i,j,2}$ as

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

Hence these two matrices define the output in terms of a three dimensional array. This array is called a tensor*.*

Multiplication of two complex numbers performed as above requires four multiplications of real numbers. In fact we can do better. Let

$$\begin{aligned} p_0 &= x_1y_1, \\ p_1 &= x_2y_2, \\ p_2 &= (x_1 + x_2)(y_1 + y_2). \end{aligned}$$

Then $z_1 = p_0 - p_1$ and $z_2 = p_2 - p_0 - p_1$ and thus three multiplications are sufficient. It can be shown (and this is one of the homework problems) that two multiplications do not suffice.

These facts turn out to be related to the *rank* of a tensor. To motivate this concept, consider the contribution of p_0 to the output, namely

$$\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} -1 & 0 \\ 0 & 0 \end{pmatrix},$$

the contribution of p_1 is

$$\begin{pmatrix} 0 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} 0 & 0 \\ 0 & -1 \end{pmatrix},$$

and for p_2, the contribution is

$$\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}.$$

Note now that the addition of these tensors gives the tensor of our problem, that is

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

Definition 11.2. *A tensor has* rank 1 *if $c_{i,j,k} = d_i e_j f_k$ for some vectors d, e and f.*

This definition carries over to ordinary matrices, since there we have that $m_{i,j} = a_i b_j$ for vectors a and b iff the rank is one.

Definition 11.3. *The* rank *of a tensor C is the least number of rank 1 tensors, N_i, such that $C = \sum_i N_i$.*

If this definition is applied to matrices we recover the ordinary concept of rank.

Theorem 11.4. *The rank of a tensor equals the number of multiplications that is required to compute the underlying computational problem.*

Proof. First, note that a tensor of rank 1 in a natural way corresponds to a computation consisting of 1 multiplication. Next, there is a one-to-one correspondence between the decomposition of the tensor into rank 1 tensors and the corresponding multiplications in the computational problem. Together this proves the theorem. □

Theorem 11.4 implies that it would be useful to be able to compute the rank of a tensor, but while computing the rank of a matrix can be achieved quickly by Gaussian elimination, for a tensor the problem is NP-complete [25].

As a curiosity we note that the rank of a tensor depends on the field over which it is computed not only on the domain of the coefficients. This is another difference to rank of matrices. The example below shows that the rank is different over $\mathbb{R}$ and over $\mathbb{C}$.

Example 11.5. *Consider the tensors*

$$\begin{pmatrix} 1 & i \\ i & -1 \end{pmatrix} \begin{pmatrix} -i & 1 \\ 1 & i \end{pmatrix}$$

and

$$\begin{pmatrix} 1 & -i \\ -i & -1 \end{pmatrix} \begin{pmatrix} i & 1 \\ 1 & -i \end{pmatrix},$$

which both have rank 1. In the first case we can choose $d = (1, i)$, $e = (1, i)$, *and* $f = (1, -i)$ *and in the second case we have* $d = (1, -i)$, $e = (1, -i)$, *and* $f = (1, i)$. *Adding these together gives the tensor for computing complex multiplication. Hence this tensor has rank 3 over* $\mathbb{R}$ *but rank 2 over* $\mathbb{C}$.

An alternative, but equivalent, way to view a tensor is as a collection of matrices, M_i. The rank is then the least number of rank 1 matrices, N_j, such that each M_i can be written as a on the form $\sum_j a_j N_j$ for some scalars a_j.

Example 11.6. *The tensor*

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$$

corresponds to

$$\begin{aligned} z_1 &= x_1 y_1 + x_2 y_2, \\ z_2 &= x_1 y_2. \end{aligned}$$

Clearly the computation of z_1 *and* z_2 *can be performed with 3 multiplications an let us consider the question of using only 2 multiplications? If this is possible the rank of the tensor must be 2, which means that the two matrices of the tensor can be written as linear combinations of two rank 1 matrices* N_1 *and* N_2. *We may suppose that* $N_1 = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$. *This follows sine if we have two other rank 1 matrices* N_1' *and* N_2 *and* $N_1 = aN_1' + bN_2$ *with* $a \neq 0$ *then we can replace* N_1' *by* $\frac{1}{a}(N_1 - bN_2)$ *giving a representation of both matrices in terms of* N_1 *and* N_2. *In this case we must have*

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = a \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} + N_2 \ \Rightarrow \ N_2 = \begin{pmatrix} 1 & -a \\ 0 & 1 \end{pmatrix}.$$

However, for all values of a, *the rank of* N_2 *is greater than 1, which implies that the given tensor can not be written as linear combination of only two rank 1 matrices. Hence, the rank of the given tensor is 3, and the computation of* z_1 *and* z_2 *then must take 3 multiplications.*

Example 11.7. *Consider again the tensor given in the previous example. For any* ε *the tensors*

$$\begin{pmatrix} 0 & \varepsilon^{-1} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 0 & \varepsilon \end{pmatrix},$$

and

$$\begin{pmatrix} 1 & -\varepsilon^{-1} \\ 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix},$$

have rank 1. Adding these together yields

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 0 & \varepsilon \end{pmatrix}.$$

So by making ε *small, we may produce a tensor arbitrarily close to the tensor of the previous example. This calls for the following definition.*

Definition 11.8. *The* border rank *of a tensor, T, is the smallest integer g, such tensors of rank t exist arbitrarily close to T.*

From Example 11.5 and 11.7 we thus have a tensor whose rank is 3 but whose border rank is 2. For ordinary matrices the border rank is always equal to the rank and hence for matrices border rank is not an interesting concept.

A final difference that we note is that the maximal rank of an $n \times n$ matrix is of course n, but the exact value of the maximal rank, r, of an $n \times n \times n$ tensor is unknown. The value of r is known only up to the inequality $n^2/3 \leq r \leq n^2/2$.

11.3 Matrix multiplication and tensors

To return to the main subject of this chapter, we want to compute

$$\begin{pmatrix} x_{1,1} & x_{1,2} \\ x_{2,1} & x_{2,2} \end{pmatrix} \begin{pmatrix} y_{1,1} & y_{1,2} \\ y_{2,1} & y_{2,2} \end{pmatrix} = \begin{pmatrix} z_{1,1} & z_{1,2} \\ z_{2,1} & z_{2,2} \end{pmatrix}$$

and now we may ask what tensor this corresponds to. We have, for example, that $z_{1,1} = x_{1,1}y_{1,1} + x_{1,2}y_{2,1}$, which can be expressed as a 4×4 matrix with rows labeled $x_{1,1}$, $x_{1,2}$, $x_{2,1}$, $x_{2,2}$, and columns labeled $y_{1,1}$, $y_{1,2}$, $y_{2,1}$, $y_{2,2}$, that is,

$$z_{1,1} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}.$$

Hence, we get the tensor

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

where the matrices correspond to $z_{1,1}$, $z_{1,2}$, $z_{2,1}$ and $z_{2,2}$, respectively.

In general we have that each entry in the matrix product is

$$z_{i,k} = \sum_{j=1}^{n} x_{i,j}y_{j,k}.$$

Note that this notation actually is a simplification, since we really have n^2 variables each of z_{k_1,k_2}, x_{i_1,i_2}, and y_{j_1,j_2}, and hence we should write the bilinear problem as

$$z_{k_1,k_2} = \sum_{i_1,i_2,j_1,j_2,k_1,k_2} c_{(i_1,i_2),(j_1,j_2),(k_1,k_2)} x_{i_1,i_2} y_{j_1,j_2},$$

where $c_{(i_1,i_2),(j_1,j_2),(k_1,k_2)} = 1$ if $i_1 = k_1$, $j_1 = i_2$, and $k_2 = j_2$, and 0 otherwise. For example, for $z_{1,1}$ we get $c_{(1,i_2),(j_1,1),(1,1)} = 1$ whenever $i_2 = j_1$, as we expect.

We find it convenient to express our tensor as the formal sum

$$\sum_{i,j,k} x_{i,j}y_{j,k}v_{i,k},$$

that is, instead of having the variables $z_{i,k}$ as output we introduce formal variables $v_{i,k}$ and move them into the sum. (As an analogy, consider a vector $(a_0, a_1, \ldots, a_{n-1})$ which can be written as a formal power series $a_0 + a_1 t + \cdots + a_{n-1}t^{n-1}$.) Note that below the formal variables are called $z_{i,k}$, changing the meaning of this notation.

Definition 11.9. *Let $T(m,p,r)$ denote the tensor that corresponds to the multiplication of an $m \times p$ matrix with a $p \times r$ matrix, that is,*

$$T(m,p,r) = \sum_{i=1}^{m}\sum_{j=1}^{p}\sum_{k=1}^{r} x_{i,j}y_{j,k}z_{i,k},$$

and let $M(m,p,r)$ denote the number of multiplications that this matrix multiplication requires, that is, $M(m,p,r)$ is the rank of $T(m,p,r)$ (For our original problem, we are hence interested in knowing the value of $M(n,n,n)$.)

Theorem 11.10. *We have that $M(m,p,r) = M(p,r,m)$, and likewise for all permutations of m, p, and r.*

Proof. Since

$$T(m,p,r) = \sum_{i=1}^{m}\sum_{j=1}^{p}\sum_{k=1}^{r} x_{i,j}y_{j,k}z_{i,k},$$

and

$$T(p,r,m) = \sum_{j=1}^{p}\sum_{k=1}^{r}\sum_{i=1}^{m} x_{j,k}y_{k,i}z_{j,i},$$

we see that $T(m,p,r)$ and $T(p,r,m)$ are equal up to a permutation of the indices and this permutation does not change the rank. □

The proof of theorem 11.10 might seem obvious but we have in fact done something. Try to prove with resorting to our present formalism that computing the product of a 4×3 matrix and a 3×2 matrix requires exactly as many multiplications as computing the product of a 3×4 matrix and a 4×2 matrix! The idea to stop distinguishing inputs and outputs is surprisingly powerful.

Definition 11.11. *Let T_1 and T_2 be two tensors which can be written as $c^1_{i_1,j_1,k_1}$ and $c^2_{i_2,j_2,k_2}$, respectively, where $i_1 \in I_1$, $j_1 \in J_1$, $k_1 \in K_1$, $i_2 \in I_2$, $j_2 \in J_2$, and $k_2 \in K_2$. The* tensor product, *$T_1 \otimes T_2$, is a tensor with index sets $(i_1,i_2) \in I_1 \times I_2$, $(j_1,j_2) \in J_1 \times J_2$, and $(k_1,k_2) \in K_1 \times K_2$, such that $c_{(i_1,i_2),(j_1,j_2),(k_1,k_2)} = c^1_{i_1,j_1,k_1} \cdot c^2_{i_2,j_2,k_2}$.*

Theorem 11.12. *For two tensors, T_1 and T_2, we have that*

$$\mathrm{rank}(T_1 \otimes T_2) \leq \mathrm{rank}(T_1)\mathrm{rank}(T_2).$$

Proof. Let $r_1 = \mathrm{rank}(T_1)$, that is, we have

$$T_1 = \sum_{i=1}^{r_1} N_i^1,$$

where each N_i^1 is a tensor of rank 1. Correspondingly, for T_2 we have $T_2 = \sum_{j=1}^{r_2} N_j^2$. This means that

$$T_1 \otimes T_2 = \sum_{i=1}^{r_1} N_i^1 \otimes \sum_{j=1}^{r_2} N_j^2 = \sum_{i=1}^{r_1} \sum_{j=1}^{r_2} N_i^1 \otimes N_j^2,$$

since $\otimes$ is distributive over $+$. It is easy to check that $N_i^1 \otimes N_j^2$ is a tensor of rank 1, and hence, the theorem follows. $\square$

Theorem 11.13. *For two tensors, $T(m_1, p_1, r_1)$ and $T(m_2, p_2, r_2)$, we have that*

$$T(m_1, p_1, r_1) \otimes T(m_2, p_2, r_2) = T(m_1 m_2, p_1 p_2, r_1 r_2).$$

Proof. Let

$$T(m_1, p_1, r_1) = \sum_{i_1=1}^{m_1} \sum_{j_1=1}^{p_1} \sum_{k_1=1}^{r_1} x_{i_1,j_1} y_{j_1,k_1} z_{i_1,k_1},$$

and

$$T(m_2, p_2, r_2) = \sum_{i_2=1}^{m_2} \sum_{j_2=1}^{p_2} \sum_{k_2=1}^{r_2} x_{i_2,j_2} y_{j_2,k_2} z_{i_2,k_2}.$$

Then

$$T(m_1, p_1, r_1) \otimes T(m_2, p_2, r_2) \quad = \quad \sum_{i_1,i_2} \sum_{j_1,j_2} \sum_{k_1,k_2} x_{(i_1 i_2, j_1 j_2)} y_{(j_1 j_2, k_1 k_2)} z_{(i_1 i_2, k_1 k_2)}.$$

Inspection shows that this tensor is exactly $T(m_1 m_2, p_1 p_2, r_1 r_2)$ and the proof is complete. $\square$

Using the notation we have introduced, and the two theorems above, we can state Strassen's result as follows. Suppose n is even. Then Theorem 11.13 implies that $T(n, n, n) = T(n/2, n/2, n/2) \otimes T(2, 2, 2)$ and Theorem 11.12 says that $M(n, n, n) \leq M(n/2, n/2, n/2) M(2, 2, 2) = M(n/2, n/2, n/2) \cdot 7$. By induction we get that if $n = 2^k$ then $M(n, n, n) \leq 7^k$ and this is precisely what we get by Strassen's algorithm.

11.4 The Complexity of Matrix Multiplication

$M(n, n, n)$ grows like n^w for some w. We are interested in the value of the exponent w. We find and describe an algorithm that gives an upper bound on the size of w. The exponent w corresponding to a certain algorithm is called the *efficiency* of the algorithm. We need also to define the efficiency of algorithms doing many disjoint matrix multiplications and also of algorithms doing only approximate matrix multiplication. The key to the current algorithm is to prove that we can convert such algorithms to exact algorithms for one matrix multiplication without decreasing the efficiency, except by an arbitrarily small amount.

What we try to find is thus:

$$w = \inf \frac{\log M(n, n, n)}{\log n}.$$

The number of multiplications needed to multiply an m by p matrix by a p by r matrix is denoted by $M(m,p,r)$ and we claim that

$$M(m,p,r) \propto (mpr)^{w/3}$$

which gives

$$w = \inf 3\frac{\log M(m,p,r)}{\log(mpr)}.$$

To see that this defines the same number note that this second definition can only give a smaller number since we have extended the range of the infimum. On the other hand know we from before that $M(mpr,mpr,mpr) \leq M(m,p,r)^3$ and this gives the reverse inequality.

Now it is time for the main idea of this chapter. It is to somehow try to make more than one matrix multiplication at the same time. We need to define also the efficiency of such an algorithm. If we compute the product of t disjoint matrices, where the i'th problem is of size (m_i, p_i, r_i), by, in total, Q multiplications, then the efficiency of that algorithm is defined to be the number w such that

$$\sum_{i=1}^{t}(m_i p_i l_i)^{w/3} = Q.$$

In our case we want to do two multiplications, one consisting of matrices with elements x_{ij} and y_{jk} which should give the elements z_{ik} as answer, and similarly one with u_{jk} and v_{ki} giving w_{ji} as answer. The indices are in the intervals $i \in [1,m]$, $j \in [1,p]$ and $k \in [1,r]$. We use the following equations.

$$\sum_j (x_{ij}+u_{jk})(y_{jk}+\varepsilon v_{ki}) - \sum_j u_{jk}y_{jk} - \varepsilon v_{ki}\sum_j (x_{ij}+u_{jk}) = \sum_j x_{ij}y_{jk} = z_{ik}$$

$$\sum_k (x_{ij}+u_{jk})(y_{jk}+\varepsilon v_{ki}) - \sum_k u_{jk}y_{jk} - x_{ij}\sum_k (y_{jk}+\varepsilon v_{ki}) = \varepsilon\sum_k u_{jk}v_{ki} = \varepsilon w_{ji}$$

The first equation is done for all i and k, and the second for all i and j. They represent one $T(m,p,r)$ and one $T(p,r,m)$. Since the first two terms consist of sums of the same products in both equations, these products have only to be evaluated once making the number of multiplications needed $mpr+pr+mr+pm$. Although this is already less than $2mpr$, we can do even better. Remove the third term in the first equation. This gives only an approximative answer, but it is done in $mpr+pr+pm$ multiplications.

There are two problems with this approach. First, we now have only an approximative algorithm for multiplication, and second, we can do two multiplications but have in the original problem only one. We next show that we can eliminate both problems without decreasing the efficiency except by an arbitrarily small amount. For simplicity we treat the two problems separately from each other, although it is possible to do them at the same time.

11.5 Exact multiplication with the approximative algorithm

We have an approximative algorithm that requires Q multiplications to compute $T(m,p,r)$. We convert this to an exact algorithm computing a different matrix product without loosing too much in efficiency.

Note that the approximative algorithm gives us a polynomial in ε, $P(\varepsilon)$. In our case above it has degree $d = 1$. By running the algorithm twice, with *e. g.* $\varepsilon = 1$ and $\varepsilon = 2$, we can then extrapolate the two answers to get the required $P(0)$. In general, if we have an approximative algorithm that gives us a result that is a polynomial $P(\varepsilon)$ of degree d, we can run it for $d+1$ different values of ε and then extrapolate to get the exact answer. This extrapolation is only a linear combination of the individual results, $P(0) = \sum_{i=1}^{d+1} c_i P(\varepsilon_i)$, with fixed constants c_i (as soon as we have fixed all ε_i), so it requires no extra multiplications.

This means that if we have an approximative algorithm of degree d that requires Q multiplications, we can get an exact answer in $(d+1)Q$ multiplications.

So far it does not seem like a big advantage with this whole approach. We can gain about half of the multiplications if we accept an approximative answer, but then we have to run this twice instead to find out the real answer. However, the point is that we can now use the approximative algorithm recursively, and thus blow up the win in multiplications during these steps, more than we have to pay back in repeated applications of the algorithm at the top level.

Theorem 11.14. *If we have an approximative algorithm with efficiency w, then, for any $\delta > 0$, there exists an exact algorithm with efficiency $w + \delta$.*

Proof. Suppose we have an approximative algorithm with degree d for $T(m, p, r)$ that requires Q multiplications. Then we can take the s-fold product of the algorithm on itself, *i. e.* , we run it recursively s levels. This yields an algorithm (still approximative) for $T(m^s, p^s, r^s)$ in Q^s multiplications, with degree sd. This means that we have an exact algorithm for $T(m^s, p^s, r^s)$ in $(sd+1)Q^s$ multiplications, which gives the efficiency

$$w = 3\frac{\log((sd+1)Q^s)}{\log(m^s p^s r^s)} = 3\frac{\log(sd+1) + s\log(Q)}{s\log(mpr)} \leq 3\frac{\log Q}{\log(mpr)} + \delta$$

for sufficiently large s. □

We next proceed to study how to convert an algorithm computing disjoint matrix product to an algorithm computing only one matrix product without loosing too much in efficiency.

11.6 Combining a number of disjoint multiplications into one

Suppose we can calculate two disjoint matrix multiplications $T(m_1, p_1, r_1)$ and $T(m_2, p_2, r_2)$ together with Q multiplications. This operation can be thought of as a tensor $T((m_1, p_1, r_1) \oplus T(m_2, p_2, r_2))$. If we run this recursively s levels, we get $T((m_1, p_1, r_1) \oplus T(m_2, p_2, r_2))^{\otimes s}$, and thus with Q^s multiplications we have computed 2^s matrix products. These are not of the same size, and to be precise $\binom{s}{i}$ of them have size $T(m_1^i m_2^{s-i}, p_1^i p_2^{s-i}, r_1^i r_2^{s-i})$.

Thus there are two steps involved here. First we have to deal with that the multiplications have different size, and thereafter we must combine several equal-size multiplications into one large matrix multiplication.

The first problem is unexpectedly simple. Of the 2^s different sized multiplications we only use those of one size. Rather than using the size that is most abundant, we use the size that is computationally most advantageous, *i. e.* the one that maximizes

$$\binom{s}{i}(m_1p_1r_1)^{iw/3}(m_2p_2r_2)^{(s-i)w/3}.$$

Since

$$((m_1p_1r_1)^{w/3} + (m_2p_2r_2)^{w/3})^s = \sum_{i=0}^{s}\binom{s}{i}(m_1p_1r_1)^{iw/3}(m_2p_2r_2)^{(s-i)w/3},$$

we can always find an i such that

$$\binom{s}{i}(m_1p_1r_1)^{iw/3}(m_2p_2r_2)^{(s-i)w/3} \geq \frac{1}{s+1}((m_1p_1r_1)^{w/3} + (m_2p_2r_2)^{w/3})^s$$

and this is the size we choose. It is easy to see that this implies that for any given δ and for sufficiently large s, the efficiency of the algorithm has dropped by at most δ. For the special case of two disjoint matrix multiplications we have proved the following theorem:

Theorem 11.15. *If we have an exact algorithm for disjoint matrix multiplications with efficiency w, then, for any $\delta > 0$ there exist an exact algorithm with efficiency $w + \delta$ for disjoint matrix multiplication of one single size.*

Proof. We need only show how to modify the proof to deal with an arbitrary sum rather than a sum of two terms. Suppose we have an algorithm computing t disjoint matrix multiplications and using Q multiplications. If we take an s tensor power of this algorithm then we obtain t^s different matrix multiplications by Q^s multiplications. The number of different sizes that occur is, however, at most s^t and hence there is one particular size that corresponds to a fraction at least s^{-t} of the "efficiency sum". Finally, since

$$\lim_{s\to\infty}\frac{\log(s^t)}{s} = 0,$$

for sufficiently large s we have lost at most δ in the efficiency. □

Now we can do K disjoint equal-sized matrix multiplications $T(m, p, r)$ with Q multiplications. Note that the efficiency w is the efficiency per computed matrix, $Q/K = (mpr)^{w/3}$. It is time to see what efficiency we can achieve for one single matrix multiplication.

Theorem 11.16. *If we have an exact algorithm for disjoint matrix multiplications of the same size with efficiency w, then, for any $\delta > 0$ there exist an exact algorithm with efficiency $w + \delta$ that computes a single matrix multiplication.*

Proof. We have an algorithm which with Q multiplications can perform K disjoint matrix multiplications $T(m, p, r)$. If we use it recursively one level, *i. e.* we try to compute $T(m^2, p^2, r^2)$, we can now manage K of the Q recursive multiplications with only one call to the algorithm. This means that $Q/K = L$ calls (for simplicity assuming L to be an integer) suffices and since the top level invocation of the algorithm also produces K copies, we have produced K copies of $T(m^2, p^2, r^2)$ in LQ multiplications. In general, with s recursive levels we get

K copies of $T(m^s, p^s, r^s)$ in $QL^{s-1} = KL^s$ multiplications. At the top level we can discard $K-1$ copies and save only one. The efficiency is

$$w = 3\frac{\log(KL^s)}{\log(m^s p^s r^s)} = 3\frac{\log K + s\log L}{s\log(mpr)} \leq 3\frac{\log L}{\log(mpr)} + \delta$$

for sufficiently large s. (Note that the original algorithm here uses $Q/K = L$ multiplications per produced matrix, so it should say L rather than Q when comparing the efficiency of the original algorithm with the new w.)

If Q/K is not an integer we have to proceed as follows. Let $H = \lceil Q/K \rceil$ be the smallest integer larger than Q/K. We can then repeat the above argument with H (simply by assuming that we did more multiplications than needed). The loss in efficiency might be too large but this can be decreased by first taking a t'th power of the original algorithm producing K^t copies of $T(m^t, p^t, r^t)$ to a cost of Q^t. If Let $H = \lceil Q^t/K^t \rceil$ and we repeat the above construction we get efficiency

$$w = 3\frac{\log(K^t H^s)}{\log(m^{st} p^{st} r^{st})} = 3\frac{t\log K + s\log H}{st\log(mpr)} \leq 3\frac{\log L}{\log(mpr)} + \delta$$

if we first chose t sufficiently large to make H a sufficiently close approximation of L^t and then s sufficiently large to decrease the influence of K^t. $\square$

11.7 Conclusions

The basic construction was to make two (approximative) matrix multiplications, corresponding to $T(m, p, r)$ and $T(p, r, m)$ using $mpr + pr + pm$ multiplications. This gives an efficiency of

$$w = 3\frac{\log((mpr + pr + pm)/2)}{\log(mpr)}$$

The choice of m, p, and r that gives best efficiency is $m = 7$, $p = 1$ and $r = 7$ yielding

$$w = 3\frac{\log((49 + 7 + 7)/2)}{\log(49)} = 3\frac{\log 31.5}{\log 49} \approx 2.659.$$

Our constructions now yield an exact algorithm for a single matrix multiplication, with efficiency arbitrarily close to 2.659 .

As already noted, the currently fastest known algorithm has an efficiency of 2.376. The best lower bound is given by the fact that $M(n, n, n) \geq 2n^2 - 1$ and thus we do not even know that $w > 2$. Furthermore, some researchers in the area, when forced to guess the value of w, pick the value 2.

Chapter 12

Maximum Flow

12.1 Introduction

A network is defined as a graph with a set of vertices V, a set of edges E and a capacity function $c : E \to Z^+$. In the problem of maximum flow we have two special vertices, the source s and the sink t. The problem is to find the maximum throughput, or the maximum flow, from s to t.

There are several real applications where one encounters this problem. Some examples are road and railway traffic, electrical and computer networks.

In order to complicate things further, there may be a cost function associated with each edge. A secondary condition might be to minimize the cost or the cost/throughput ratio. We her only address the basic problem.

There are often several different flows, even infinitely many, which all attain the maximum value. However, we will prove later that there is always a maximum flow with integer only values.

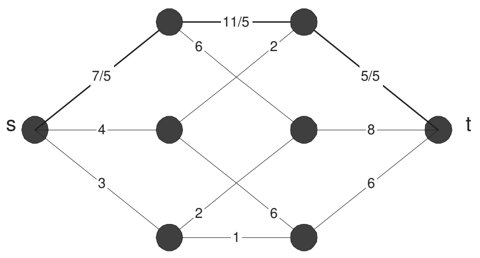

Figure 12.1: Partially filled network.

One way of finding the maximum flow is to start with some flow and increase it. An algorithm which finds an increase to an existing flow can then be run until no further improvement is possible. In Figure 12.1 we see a network with capacities on each edge. The second number on each edge is the current flow along that edge. Edges with only one number has 0 flow.

The algorithms we study require the use of the *residual graph*. A residual graph is constructed from a (partially) filled network by using the same vertices as in our original graph, and by replacing every edge with two directed edges, pointing in opposite directions. The edges are associated with a capacity, which

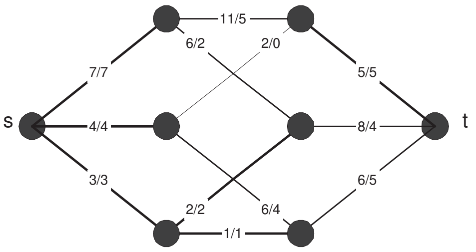

Figure 12.2: Network with increased (maximum) flow.

equals the possible flow increase in that direction. Edges in the residual graph with zero capacity are removed. The correspond to fully used edges.

For example, consider the residual graph of the flow in Figure 12.2 displayed in Figure 12.3.

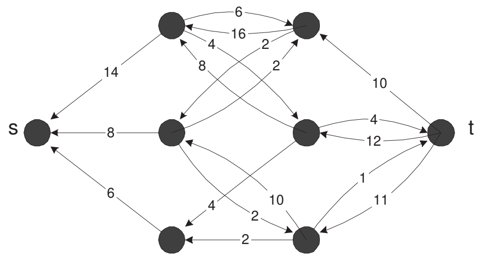

Figure 12.3: Residual graph.

At a closer study, the graph above shows an interesting property: There is no directed path from the source to the sink. This implies that it is impossible to increase the flow and it seems likely that we have found a maximum flow. This intuition is true and can be formed into of theorem.

Theorem 12.1. *If there is no directed path from s to t in the residual graph, we have a maximum flow.*

Proof. Consider the residual graph again.

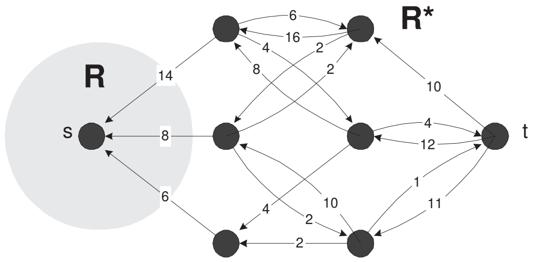

Figure 12.4: Residual graph divided into two sets, R and R^*.

Let R be the set of vertices for which a directed path from s can be found, and R^* its complement. All edges in the original graph have maximum flow directed from R to R^*. There is a certain flow from s to t. We make some of observations:

1. The flow from R to R^* equals the flow from s to t.
2. The flow from R to R^* can never exceed the total capacity of all edges $(v, w) : v \in R$ and $w \in R^*$.
3. Since the flow from R to R^* is maximized, so is the flow from s to t.

This concludes the proof. □

12.2 Naive algorithm

To find the maximum flow we start with any flow (in particular no flow at all) and, given the residual graph, find an increase of our current flow. A naive approach, where we try to find any directed path from s to t and fill it up to maximum capacity, runs for at most $O(|V|c_{max})$ iterations, since we increase the flow with at least one each iteration. A path which increases the flow is called an *augmenting path*. This search can be implemented as a depth-first search, which takes $O(|E|)$. Total elapsed time is $O(|E||V|c_{max})$, which is acceptable except when c_{max} is large. This worst case scenario is possible as can be seen in the following example.

Example 12.2. *Consider the sequence of flows given in figure 12.5.*

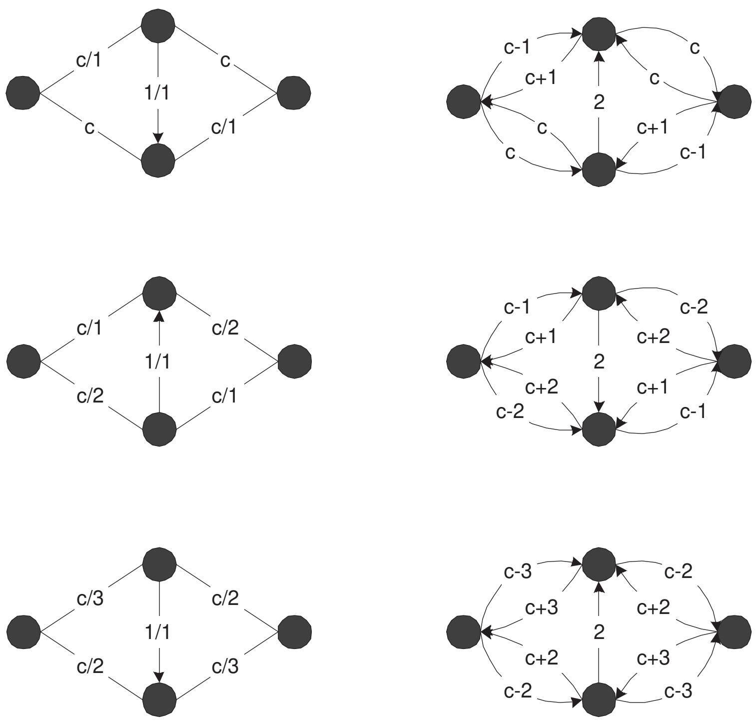

Figure 12.5: Worst case scenario.

We have lined up three partially filled network graphs and their residual graphs. In the first picture we show a graph with minimal flow. In its residual graph we find an augmenting path which gives the second graph. Its residual graph gives the third graph, which looks like the first one. It is clear that with this graph, the total flow is increased by exactly two in each step, not counting the first and the last. Thus the algorithm continues for about $c/2$ iterations.

Let us give the promised argument showing that we always have a maximal flow with integral flows along each edge. Each augmenting path gives an integer increase to our flow. We start with zero flow on each edge and hence the final, maximal flow has an integral flow along each edge.

If capacities are large, our naive algorithm can, as shown by example 12.2, be very inefficient. It turns out that we can do better by choosing an augmenting path in a clever way. When considering methods for choosing augmenting paths there are two key properties to look for.

1. It should be easy to find an augmenting path of the desired type.
2. We should get a strong bound on the number of iterations.

In all our augmenting path algorithms we always augment the flow along the found path by the maximal amount possible and hence we saturate at least one edge along the path.

12.3 Maximum augmenting path

A natural method is find the path which maximizes the increase of the flow. We can find this quickly by a bottom-up approach: Find the maximum flow $C(v)$ from s to each vertex $v \in V$ by using already calculated values of C. The algorithm can be written this way:

We want to calculate $C(v), \forall v \in V$. $G(v)$ contains a preliminary estimate.

1. $S = \{s\}$
2. $G(v) = cap(s, v)$
3. Pick the $v \in S^*$ with the largest $G(v)$.
4. $C(v) = G(v), G(w) = max(G(w),\ min(G(v), cap(v, w))), \forall (v, w) \in E$

This is essentially Dijkstra's algorithm for finding the minimum distance between vertices in a graph. For more detail consult a book on algorithms, for example [13] p.529-530.

When we have a sparse graph, $G(v)$ can be stored in a heap for which the maximum value can be found in $O(1)$, but updating costs $O(\log |V|)$. The total cost is $O(|E|log|V|)$, since there is at most one update for every edge in the fourth step in our algorithm. In a dense graph, it is cheaper to look at all the vertices in each step, giving the cost $O(|V|^2)$. Thus, the total complexity is $O(min(|E| \log |V|, |V|^2))$.

Thus we can find that path which allows the largest increase in flow efficiently and we now address the question to bound the number of iterations. To get a feeling for this problem, let us start by an example.

Example 12.3. *Consider a network with 4 levels. On level 1 there is only s and on level 4 there is only t. There are m nodes on each of the levels 2 and 3. The node s is connected to each level 2 node by an edge of capacity m and t is connected to each level 3 node also by an edge of capacity m. Every level 2 node is connected to every level 3 node by an edge of capacity 1. The reader may verify that if each edge is saturated we have flow m^2 while any augmenting path has flow 1.*

Next we show that this example is the worst possible.

Theorem 12.4. *Given a graph with maximal flow M, then there is always a path with flow $\geq \frac{M}{|E|}$.*

Proof. Suppose we have a network with a maximum flow M. Take any path from s to t. If this path has a flow $\geq \frac{M}{|E|}$, we've found such a flow. If the flow is smaller, we can remove it from the network and find another path. If the new path also has a flow $< \frac{M}{|E|}$, we remove it, find another and so on. The maximum number of times a path can be removed is $|E|$, since we reduce the flow on (at least) one edge to zero in each iteration. This, however, can't occur since all flow must be removed, and we remove less than $\frac{M}{|E|}$ each time. Therefore, there must be a path with flow $\geq \frac{M}{|E|}$. □

Given a network and a residual graph with maximum flow M, we know that there is an augmenting path with a flow greater than or equal to $\frac{M}{|E|}$. After our first iteration we have at least this flow in our network and the maximum flow in the corresponding residual graph is $\leq M(1 - \frac{1}{|E|})$. After k iterations we have a maximum flow $\leq M(1 - \frac{1}{|E|})^k$ in the residual graph. Since we increase the flow with at least 1 in every iteration, we are finished when the flow in the residual graph is < 1.

The maximal number of iterations can now be estimated as follows.

$$\begin{aligned}
(1 - \frac{1}{|E|})^k M \leq 1 \Rightarrow \\
k \log(1 - \frac{1}{|E|}) + \log M \leq 0 \Rightarrow \\
k > \frac{\log M}{-\log(1 - \frac{1}{|E|})} \approx |E| \log M
\end{aligned}$$

Since $O(|E| \log M) \leq O(|E| \log(|V| c_{max}))$ and each iteration is an application of Dijkstra's algorithm we get a total running time of

$$O(min(|E|^2 \log |V| \log(|V| c_{max}),\ |E||V|^2 \log(|V| c_{max}))).$$

The c_{max} factor has been reduced by a logarithm. It can be removed entirely with an even better way of choosing the augmenting path.

12.4 Shortest augmenting path

The second main approach to finding a good augmenting path is to find a path that contains as few edges as possible, *i. e.* , a shortest augmenting path. As before, we repeat this step until no further increase is possible. The crucial fact for this approach is that it turns out that the length of the shortest path never decreases. This is far from obvious since when we augment along a path some new edges might appear in the residual graph. These edges are always along the augmenting path and directed to the opposite direction of the flow we just created. Thus intuitively they should not be good to use in a short path since the augmenting path we used was as short as possible and the new edges "go the wrong way". Since we repeat such an intuitive argument is of course not sufficient and we need a formal proof.

Consider the breadth first search (BFS) tree in the residual graph and assume at this point that the distance from s to t in the residual graph is d.

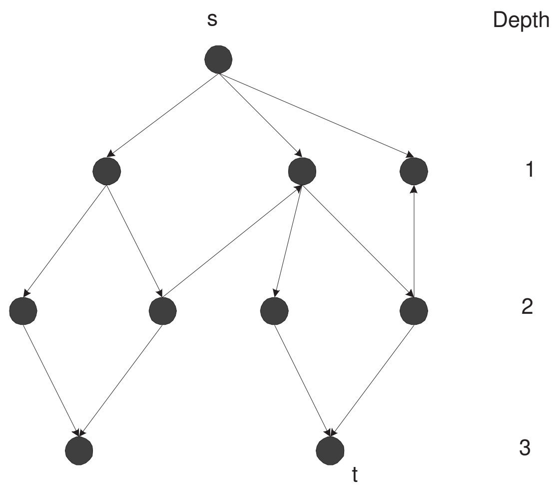

Figure 12.6: BFS tree resulting from a residual graph.

Since there are d layers (in the example $d = 3$) in the BFS-tree every path of a minimal length d must go strictly downwards in the BFS tree. New edges point upwards and since this is the only type of edge added they cannot be used for any path of length at most d from s to t. Each augmenting path saturates one edge and hence one edge pointing downward is removed in each iteration. There are hence at most $|E|$ augmenting paths of length d, after which we must choose a path of length $d + 1$. Finding a path takes $O(|E|)$, and we can find at most $|E|$ paths for each d. Since we have $|V|$ possible values of d the total running time is bounded by $O(|E|^2|V|)$.

This algorithm is good, but can be improved by finding all augmenting paths of a fixed length d more efficiently. This is natural since, as described above, they all come from a single BFS tree where we only erase edges. We proceed as follows.

1. Calculate the BFS tree.

2. Remove the edges that cannot be used to reach t in a path of length d. This is done by backtracking from t. Also remove any vertices that lose all edges connected to them.

3. Find a path from s to t of length d.
4. Calculate the flow on the found path.
5. Update the residual graph
6. Remove saturated edges in the BFS tree. Also recursively remove nodes that no longer can reach t by downward edges in the BFS tree. Goto 3.

Steps 1 and 2 use total time $O(|E|)$ for each fixed d. Steps 3,4 and 5 use $O(d)$, per path found. It is not hard to find a suitable data structure with pointers to make step 6 take $O(|E|)$ total time. We simply make sure that for each edge removed we use constant time. Total time spent for a certain depth d is thus $O(d[\text{number of paths found}] + |E|) \leq O(d|E|) \leq O(|E||V|)$. Since d can take $|V|$ different values, the total running time is $O(|E||V|^2)$, which is better than the previous algorithm since the number of vertices is almost less always than the number of edges.

We are satisfied with this result, but note that better algorithms are available. Since the most recent results depend in a subtle way on the relation between $|V|$, $|E|$ and the maximal capacity we refer to [28] for a discussion of the best results.

Chapter 13

Bipartite matching

A graph G is *bipartite* is one can divide the vertices into two sets V_1 and V_2 such that each edge is between one vertex in V_1 and one vertex in V_2. A *matching* in a bipartite graph G is a subset M of the edges E such that no two edges in M have a common vertex. A matching is a *maximum matching* for the graph if no other matching has a greater number of edges.

The problem of finding a maximum matching in a given bipartite graph is a special case of the network flow problem. To see this, we add two vertices s and t to the graph, join s to every vertex in V_1 by an edge, and join every vertex in V_2 to t (see Figure 13.2). Every edge is assigned the capacity 1. We now find a maximum flow through the network with integer flow through all edges. Then the edges with flow 1 in E form a maximum matching.

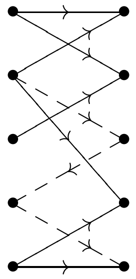

Figure 13.1: A possible augmenting path (dashed).

Consider the flow after the first step of the algorithm. This flow corresponds to a matching with one edge e. The edge in this matching is now directed the opposite way in the residual graph. Now take an augmenting path through e, and disregard the edges at s and t. We get a path where the edges alternately belong to the new matching or to the original matching (see Figure 13.1). The edges that belong to the old matching are precisely the edges that are directed from V_2 to V_1. The first and last edges in the path belong to the new matching.

Which network flow algorithm should we use? Since the maximum edge capacity is 1, the naive algorithm runs in time $O(nm)$, where $n = |V_1 \cup V_2|$ and $m = |E|$. We now analyze the $O(n^2m)$ algorithm which selects the *shortest augmenting path*, in order to find out if it performs better in the special case of flow instances that come from bipartite matching problems.

Theorem 13.1. *Let M be any matching, and let a maximum matching have $|M| + k$ edges. Then M has at least k disjoint augmenting paths.*

Proof. Let M' be a maximum matching, and consider the two matchings M and M' together (see Figure 13.3). We obtain a graph where each vertex has degree at most 2. This gives rise to three types of connected components:

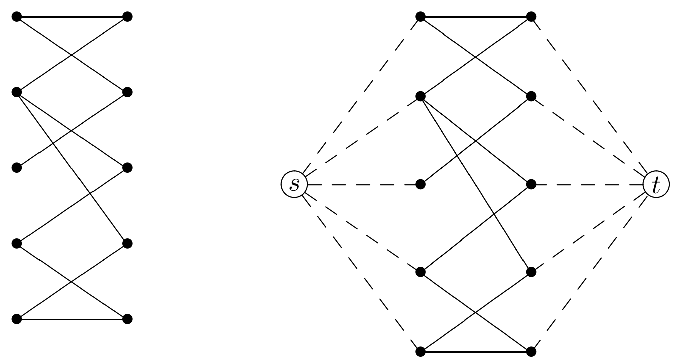

Figure 13.2: A bipartite graph and the corresponding network.

1. A non-augmenting path (or possibly a closed circuit) with an equal number of edges from M and M'.
2. An augmenting path where the first and the last edge belongs to M'.
3. A diminishing path which begins and ends with an edge from M.

There is one more edge from M' than M in an augmenting path, and one edge less in a diminishing path. Hence

$$(\text{number of augmenting paths}) - (\text{number of diminishing paths}) = k,$$

and so there must be at least k augmenting paths. □

Corollary 13.2. *If k edges can be added to the matching then there is an augmenting path of length at most n/k.*

Corollary 13.3. *If there are no augmenting paths of length $\leq \sqrt{n}$, then there are at most $\sqrt{n}$ augmenting paths remaining before the maximum matching is found.*

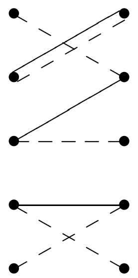

Figure 13.3: A matching with 3 edges and a maximum matching (dashed) with 5 edges.

The network flow algorithm which finds the shortest augmenting path in each step finds all augmenting paths of a fixed length d in time

$$O(d \cdot (\text{number of paths}) + m).$$

We run the algorithm until $d \geq \sqrt{n}$. This takes time $O(n\sqrt{n}+m\sqrt{n})$. We then find the remaining augmenting paths using depth-first search. The total time of this second stage is bounded by $O(m\sqrt{n})$. We may assume that $m \geq n$, and hence the total running time for the algorithm is bounded by $O(m\sqrt{n})$. This is the best algorithm known for the bipartite matching problem. We summarize our findings in a theorem.

Theorem 13.4. *A maximal bipartite matching in an unweighted graph can be found in time $O(|E|\sqrt{|V|})$.*

We now complicate the problem by adding weights to the edges and asking for the minimal weight, maximal size matching.

13.1 Weighted bipartite matching

13.1.1 Introduction

Suppose that we have a bipartite graph $(V_1 \cup V_2, E)$ and a weight function $w{:}E \to \mathbb{Z}$. The weight of an edge may be thought of as the cost for including the edge in a matching. The goal is now to find a matching of maximal size, and among these, the "cheapest" one. That is, we must find a maximum matching M with $\sum_{e \in M} w(e)$ minimal.

Note that it does not matter if some of the weights are negative; if we increase all weights by an equal amount, the solution is preserved, since all maximum matchings have the same number of edges. We can therefore assume that $w(e) \geq 0$ for all $e \in E$.

The key idea is to find the cheapest matching with k edges, for successive k. The following theorem shows that we can always do this by choosing the *cheapest* augmenting path, that is, the path that causes the smallest increase in the total weight of the matching.

Theorem 13.5. *Let M be a cheapest matching with k edges. A cheapest matching with $k+1$ edges can be obtained by augmenting M with the cheapest augmenting path.*

Proof. Let M' be a cheapest matching with $k+1$ edges, and consider the bipartite graph H with edge set $M \cup M'$. The situation is analogous to that in the proof of Theorem 13.1. Specifically, we get the same types of connected components:

1. Non-augmenting paths with an even number of edges.

2. Augmenting paths with one edge more from M' than M.

3. Diminishing paths with one edge more from M than M'.

In a non-augmenting path, the total weight of edges in M must be equal to the total weight of edges in M'. Otherwise one of the matchings could be made cheaper by replacing the edges in the path by those in the other matching.

The number of augmenting paths is one greater than the number of diminishing paths. If there are no diminishing paths, then we are done. Otherwise, consider a pair consisting of one augmenting path and one diminishing path. The total weight of the edges in both paths that belong to M must equal the weight of the edges in M' in the two paths. Otherwise we could switch the edges in M and M' and obtain a cheaper matching of size k or $k+1$. It follows that *all* augmenting paths have the same difference c between the weight of edges in M' and M, while all diminishing paths have weight difference $-c$, where c is a non-negative number. Thus any augmenting path is equally cheap.

It follows that we always get a cheapest matching with $k+1$ edges by augmenting M with the cheapest augmenting path. $\square$

The problem of finding the cheapest augmenting path is similar to the shortest-path problem, Since the edges belonging to M are removed if they are used in the augmenting path, their weights should be negated. This creates negative distances in the graph and we cannot use Dijkstra's algorithm directly. We have two options:

- Use an algorithm for the shortest-path problem that allows negative distances.

- Find a way to eliminate the negative weights.

13.1.2 The general shortest-path problem

A solution to the shortest-path problem, when negative distances are allowed, exists if and only if there are no circuits with negative total distance in the graph. A shortest *simple*[1] path always exists, but finding it in the general case is NP-complete. In our case, however, there cannot be any negative circuits, since this would contradict the fact that we have a cheapest matching.

In Dijkstra's algorithm (for details see [13]), we "guess" the shortest distance between s and the other vertices. The lowest guess for a vertex is always the correct value. This is not true if there are negative distances, but a similar algorithm works:

1. Let $D(s) = 0$ and $D(v) = \infty$ for all vertices $v \neq s$.

2. For all vertices v_i and all edges (v_i, v_j), put

$$D(v_j) = \min(D(v_j), D(v_i) + d_{ij}),$$

   where d_{ij} is the distance of the edge.

3. Repeat step 2 sufficiently many times.

The key to the analysis is to estimate the number of iterations. The answer is given below.

Theorem 13.6. *After k iterations of step 2 in the above algorithm, $D(v)$ is the length of the path from s to v which uses at most k edges.*

Proof. The theorem is proved by inductions over k and we leave the details to the reader. □

In a graph with no negative cycles, each shortest path is of length at most n and hence n iterations are sufficient. Since each iteration runs in time $O(m)$ we can find the shortest path to each vertex in time $O(nm)$. This can be compared to Dijkstra's algorithm which runs in time at most $O(m \log n)$.

[1] A path is simple if it contains each vertex at most once.

13.1.3 Removing the negative distances

We now try to scale the weights to make them positive. This can be done by finding an appropriate function $F\colon V \to \mathbb{Z}$ and setting

$$w'_{ij} = w_{ij} + F(v_i) - F(v_j) \tag{13.1}$$

for all edges (v_i, v_j) with original weights w_{ij}. This changes the total cost of any path from s to t by $F(s) - F(t)$. Since this change does not depend on the path chosen, finding the path with lowest weight is the same problem independent whether we use the weights w_{ij} or w'_{ij}.

One particularly interesting choice is to take $F(v)$ as the cost of the cheapest path from s to v. Then the inequality $F(v_j) \leq F(v_i) + w_{ij}$ holds, and the new weights w'_{ij} are non-negative. This might seem like a circular argument since we say that once we have the shortest distances we can modify the weights to make them all positive. The reason we wanted positive weights in the first place was to find these shortest distances! It turns out, however, that we can enter the "circle" and stay there. The weights are intially positive and hence we can find $F(v)$ in the first iteration. We update the weights with this $F(v)$ using (13.1). This change makes any edge that appears on the shortest path from s to t have weight 0 and since these are the only edges that get reversed, changing the sign of those weights does not create any negative numbers and thus we can continue. The algorithm is thus follows:

1. Find the cheapest path from s to t using Dijkstra's algorithm. The distances $F(v)$ of all vertices are computed at the same time. If no such path exists, the algorithm terminates.
2. Modify the weights on all edges. Weights w_{ij} are replaced by the w'_{ij} using (13.1).
3. Add the augmenting path, reversing the directions of the edges along the path. Go back to step 1 and repeat.

This algorithm requires at most n iterations of Dijkstra's algorithm. The running time is therefore $O(mn \log n)$. We state this as a theorem.

Theorem 13.7. *A minimal weight maximal matching in a bipartite graph can be found in time $O(|E||V| \log |V|)$.*

Figures 13.4 to 13.11 illustrate the algorithm.

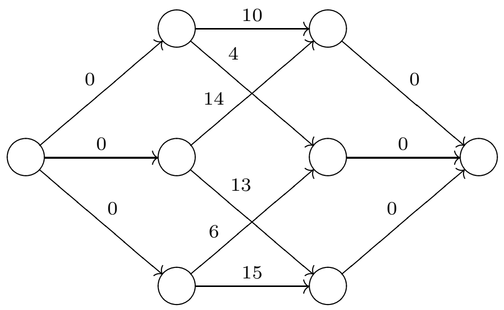

Figure 13.4: The algorithm for weighted matching. The initial situation.

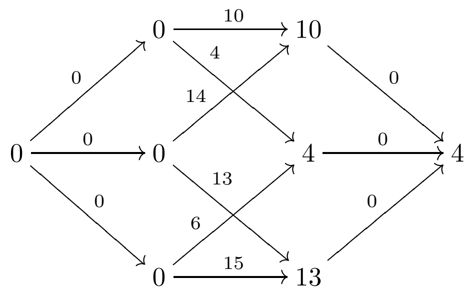

Figure 13.5: After Dijkstra's algorithm.

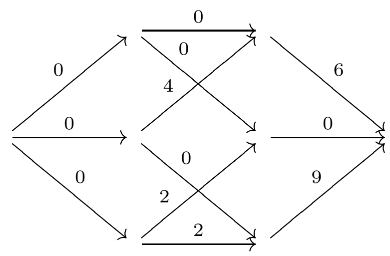

Figure 13.6: Modified weights.

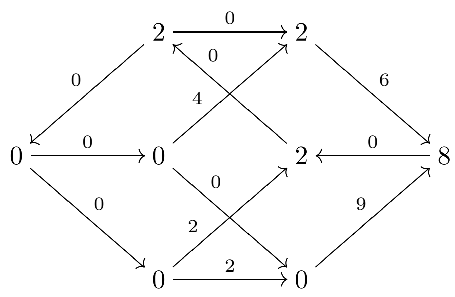

Figure 13.7: Augmenting path added, edges reversed and vertices relabeled with the new shortest distances.

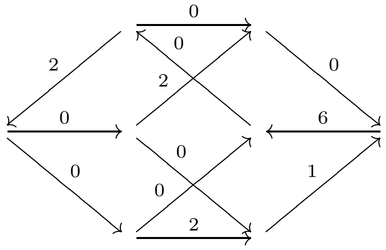

Figure 13.8: Modified weights.

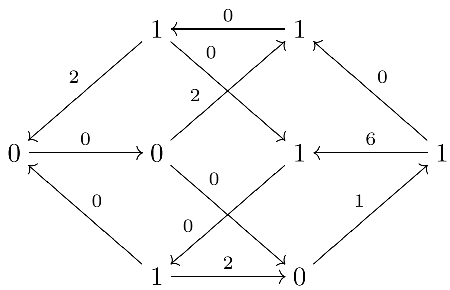

Figure 13.9: Second augmenting path added.

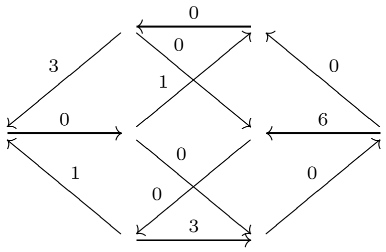

Figure 13.10: Modified weights.

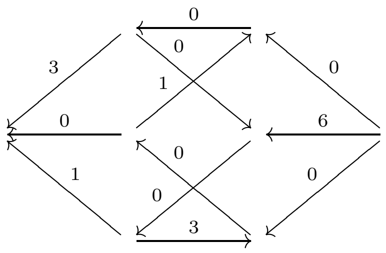

Figure 13.11: Third augmenting path added. Since t cannot be reached from s, we are done. The matching consists of the reversed edges.

Chapter 14

Linear programming

14.1 Introduction

A linear programming problem is a special case of a mathematical programming problem. One general class of problems is described as follows: Find

$$\min f(x)$$

given the constraints

$$h_i(x) \leq 0 \quad i = 1, 2, \ldots m,$$

where f and h_i are some type of functions. It does not matter if we use minimization or maximization since finding $\min f(x)$ is equivalent to finding $\max -f(x)$. In these notes we allow both maximization and minimization problems. The function f is called the *objective function* while the functions h_i are the *constraints*. In linear programming, both the objective function and the constrains are linear, and we formulate the problem as

$$\min c^T x$$

$$Ax \leq b,$$

where c, b and x are column-vectors and A is a matrix. We assume that we have n variables and m inequalities and thus c and x are of length n, b is of length m and A has m rows and n columns.

14.1.1 Why Linear Programming?

From an applications perspective, linear programming is a very useful subroutine and sometimes a problem can be formulated precisely as a linear program. As a, somewhat contrived, example of the latter case consider a situation where we have m jobs to be performed and k workers (or machines). We have km variables x_{ij}, one for each combination of task j and worker i. It is supposed to model the amount of effort of worker i invests in task j.

$\sum_{j=1}^{m} x_{ij} \leq c_i$, the amount of time person i is willing to work.

$\sum_{i=1}^{k} x_{ij} w_{ij} \geq 1$, every task is supposed to be completed.

Here the weight factors w_{ij} give the speed of worker i on task j. Given these constraints we want to minimize the total cost which we formulate as $\sum_{i,j} x_{ij} l_i$ where l_i is the salary of person i. Note that all constraints as well as the objective functions are linear.

A common way to use linear programming for more complicated tasks is through integer linear programming. Such a problem is obtained by adding the constraint that the variables take only integer values. Unfortunately, it is not difficult to see that integer linear programming is NP-hard and thus this formulation might be of little value. It turns out, however, that this is, in many situation not correct. A common approach is to simply drop the integrality constraint and solve the linear program. If we are lucky the solution is integral and if we are not, depending on the situation, the solution might still give useful information. In particular since dropping the integrality constraint can only improve the optimum the optimum over that rational numbers is a bound for the optimum of the integral problem. One major industrial use of this approach is the scheduling of air-line crews.

We now turn to studying the problem.

14.1.2 Example 1

Find the maximum for $x + 2y$ given the following conditions

$$\begin{aligned}
&x - y \leq 2 \\
&2x + y \leq 16 \\
&y \leq 10 \\
&x, y \geq 0
\end{aligned}$$

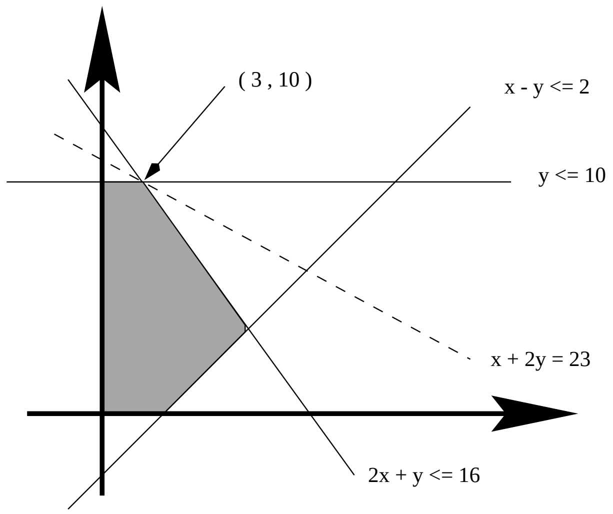

Figure 14.1: *Example 1*

We denote the set of points which satisfy the constraints by K and it is the area shaded grey in Figure 14.1. Note that K is convex, which is not surprising since it the intersection of halfspaces which are convex and the intersection of any collection of convex sets is convex. Remember that a set K is convex if for $x, y \in K$ it is true that the line segment between x and y also belongs to K. In mathematical terms we have $\lambda x + (1-\lambda)y \in K, 0 \leq \lambda \leq 1$.

A set K which is the intersection of halfspaces is called a *polytope*. The geometry of such a body is important and let us define a few terms. We assume that the hyperplanes defining K are in general position, by which mean that any n intersect at a single point and any $n-1$ intersect along a line and no set of of $n+1$ hyperplanes have a common intersection. The *interior* of K is the set of points such that we have strict inequality in all the constraints, this implies in particular that a small ball around any such point is still contained in K. A *vertex* of K is a point that is the intersection of n defining hyperplanes and which satisfies the other inequalities strictly. Finally we let an *edge* be the set of points lying in $n-1$ of hyperplanes and satisfying the other inequalities strictly. To avoid discussing degenerate cases we also assume that K is bounded.

In the example in Figure 14.1 it easy to see that we have a maximum at the point (3,10). We note that this time the optimum is in a vertex of K. It turns out that this is always true in that any linear program has an optimal point which is a vertex. The optimum might also be obtained at non-vertices but in such a case the optimum is non-unique and some vertex is an acceptable answer. That this is true is a general property of convex functions over convex sets and we will not give the formal proof, but only a short motivation. If v is an optimal point and it is not a vertex, then it can be written on the form $\lambda x + (1-\lambda)y$ with $x, y \in K$ and $0 < \lambda < 1$. It is not difficult to see that both x and y must attain the same value and thus we can find an optimal point closer to the boundary and if we continue this way we eventually end up with a vertex.

14.1.3 Naive solution

Given that the optimum is always at a vertex, one way to solve the problem is simply to check all vertices, and take the vertex that results in the best value for the objective function. There are potentially $\binom{m}{n}$ ways to chose the n equations that define a vertex and this is an exponentially large number. Most of these sets of n equation will define a point outside K, but a simple example that shows that we might still end up with exponentially many is given by the hypercube defined by

$$0 \leq x_i \leq 1, \ i = 1, \ldots, n.$$

In this case $m = 2n$ and we have 2^n vertices.

14.2 Feasible points versus optimality

A feasible point is any point that satisfies the given inequalities and thus belongs to K. Before we attack the question of how to find the optimum let us address the question of finding *any* point in K. In fact already this is a difficult problem which is almost equivalent to finding the optimal point. This is explained by the following theorem.

Theorem 14.1. *Suppose that $K \subseteq [-M, M]^n$ is defined by m inequalities and that $\max_i |c_i| \leq C$. Then if we can solve the feasibility problem in n variables* *with $m+1$ inequalities in time T then we can find the optimal problem within accuracy ϵ in time $T \log(2nCM\epsilon^{-1})$.*

Proof. The set of inequalities we use is the inequalities defining K together with $c^T x \leq D$ for a suitable sequence of choices for D. Initially we know that the optimum is in the range $[-nCM, nCM]$. By a binary search over D we can, in t calls to the feasibility algorithm, decrease the length of the interval where the optimum is located by a factor 2^t. The result follows. □

We have a more positive theorem which in fact tells us that we can assume that we start with a feasible point.

Theorem 14.2. *Given a linear program with n variables and m inequalities. If we, starting from a feasible point, can find the optimal point in time $T(n, m)$ then we can find a feasibly point (provided it exists) for any linear program with n variables and m inequalities in time $T(n+1, m+1)$.*

Proof. We are trying to find a feasible point to the inequalities $Ax \leq b$. We the same extra variable y to each inequality transforming the i inequality to

$$\sum_{j=1}^{n} A_{ij} + y \leq b_i.$$

We also add the equation $y \leq 0$ and ask for $\max y$. It is not difficult to see that the original system is feasible iff the optimum of the created problem is 0 and that any optimal solution gives a feasible solution to the original system. Thus to complete the proof we need just find a feasible solution to the created system. This is easy since we can take any value of x and adjust y appropriately. For definiteness we can use $x = 0$ and $y = \min(b_1, b_2, \ldots b_m, 0)$. □

In view of Theorem 14.2 we can assume that we are given a feasible point and we want to find the optimum. Given any feasible point we can find a vertex as follows. Suppose that at a given point x some inequalities are satisfied with equality and this gives a submatrix A_x. If x is not already a vertex then A_x contains less than n equations and we want to modify x to add more satisfied equalities. Since A_x has fewer equations than unknowns we can then find $y \neq 0$ such that $A_x y = 0$. We can now change x to $x + ty$ for a suitable t. The inequalities originally satisfied with equality remain equalities, and we find a minimal t that gives an equality in one of the other inequalities. We pick up another equality and within n iterations we are at a vertex.

Before we embark on describing algorithms let us give the intuitive reason why linear programming should be easy. The standard method for optimization problems is that of local search. We start with some solution and create better and better solutions by locally improving the solution. If we are dealing with rational numbers this usually means some kind of steepest descent, we try to move the current solution to a better solution by following the gradient of the objective function.

The reason this method does not produce optimal solutions in general is that the method gets stuck in a local optimum which is different from the global optimum. There is no small way to improve the solution but a radical change does create a better solution. Linear programming has the good property of having no local optima. The only local optimum is the global one. Thus if we can simply keep moving in a systematic fashion we will get to the optimal solution, the only problem is to get there quickly.

14.3 Simplex Method for n dimensions

The first, and most famous algorithm for linear programming is the simplex method. It was originally suggested by Danzig in 1947 and it remained the only practical method until the middle of 1980'ies.

This method starts at a given feasible vertex and moves from vertex to vertex improving the objective function. To see how this is done, let us assume that we are at a particular vertex H. Reorder the inequalities so that equality in the first n inequalities define H and that matrix A splits into A_H (inequalities satisfied with equality at H) and the rest A_R (inequalities which are strict at H. We have

$$\begin{pmatrix} A_H \\ A_R \end{pmatrix} x \leq \begin{pmatrix} b_H \\ b_R \end{pmatrix}$$

The objective is to check if the current vertex is the optimum or to find a nearby vertex that improves the objective function. More concretely we observe that the $n \times n$-matrix A_H is invertible by the assumption that each n hyperplanes define a vertex and that $x = A_H^{-1} b_H$ gives the coordinates of H (we will from now on blur the distinction between a vertex and its coordinates). The edges leaving this vertex are obtained by relaxing one of the inequalities, i.e. replacing b_h by $b_h - te_i$ where $t > 0$ and e_i is the i'th coordinate vector. This gives

$$x = A_H^{-1} b_H - t A_H^{-1} e_i.$$

Since we are trying to maximize $c^T x$ and $t > 0$ this leads to an improvement iff $c^T A_H^{-1} e_i < 0$. If there is no such i we are at the optimum and iff there is an i we follow that edge. We replace x by $A_H^{-1} b_H - t_{max} A_H^{-1} e_i$ where t_{max} is the maximal value such that all other inequalities are true. For this value of t one of the formerly strict inequalities is now an equality. We have reached a new vertex and repeat the process. Note that we are here using the assumption that no $n + 1$ of the hyperplanes intersect in common point. This property implies that $t_{max} > 0$ and we make progress. If we have a degenerate situation with more than n hyperplanes intersecting at a point we need to be more careful to make sure we do make progress. This is important in practice, but we do not discuss this detail here. Let us summarize.

The simplex algorithm

- At a vertex H find the equalities which are satisfied with equality, forming a matrix A_H.
- Invert A_H.
- Find an i, such that $c^T A_H^{-1} e_i < 0$. If no such i then H is the optimum, output H and halt. Otherwise for one found i replace H by $H - t_{max} A_H^{-1} e_i$ where t_{max} is the maximal t such that $H - tA_H^{-1} e_i$ satisfies $Ax \leq b$.
- Repeat.

To analyze the time complexity of this algorithm, the key is to bound the number of iterations. It turns out that in the worst case this can be very large. There is a famous example of Klee and Minty that is a perturbation of the hypercube and where the algorithm in fact goes through all 2^n vertices. In practical experience, however, most people report that it is unusual with problems needing more than $3n$ iterations. Thus it seems like the worst case instances do not accurately reflect the true behavior of the algorithm. Some attempts have been made to analyze the average behavior of simplex [7], but the relevance of these results have been disputed.

A combinatorial question of some interest related to the simplex algorithm is the following. Suppose we have a graph G that is connected. The distance between any two nodes is the minimal number of edges to get from one to the other. The diameter of G is the maximal distance of any two vertices. Now the vertices of a polytope K naturally form a graph where two are connected if whey are joined by an edge in K. Suppose we have n variables and m inequalities defining K. Then any simplex type algorithm that moves from vertex to vertex will need at least as many moves as the diameter of the corresponding graph. Suppose for simplicity that $m = 2n$. Then it is conjectured that the diameter of K is bounded by cn for some absolute constant c but the strongest upper bound known is a result by Kalai and Kleitman giving the bound $m^{\log n+2}$.

14.3.1 Dual linear programs

It turns out that each linear program has a companion, the *dual* linear program. We will here only scratch the surface of the importance of this problem. When discussing the dual problem, the original problem is usually called the *primal*. Below we give the find the primal together with its dual.

Primal	Dual
$\max c^T x$	$\min b^T y$
$Ax \leq \mathrm{b}$	$A^T y = c$
	$y \geq 0$
n variables	m variables
m inequalities	n equalities
	m inequalities

The origin of the dual problem is general Lagrange duality used when solving optimization problems in many variables and let us elaborate slightly on this connection. The simplest case is to maximize $f(x)$ given $h(x) \leq 0$ for some differentiable functions f and h. Then it is not difficult to see that in the optimal point, the gradients of f and h (denoted ∇f and ∇h, respectively) are parallel. If we have a more general condition given by the inequalities $(h_i(x) \leq 0)_{i=1}^m$ the corresponding condition is that at the optimal point the gradient of f is a positive linear combination of the gradients of h_i. Furthermore if

$$\nabla f = \sum_{i=1}^{m} \lambda_i \nabla h_i$$

where $\lambda_i \geq 0$ then λ_i may be positive only if $h_i(x) = 0$ at the optimum point and thus for the optimal point we have $\lambda_i h_i(x) = 0$ for all i. In our case both f and h are linear functions, c is the gradient of f, A^T gives the gradients of the constraints and the y-variables correspond to the multipliers λ_i.

We will here establish three properties of the dual problem.

1. For arbitrary feasible points y and x of the dual and primal respectively we have $b^T y \geq c^T x$.

2. The optima of the two problems (if they exist) are equal.

3. The two problems do form a pair in that the dual problem of the dual problem is again the primal problem.

The first fact tells us that any feasible solution to the dual problem gives an upper bounds on the optimum of the primal. The second property tells us that this bound can be arbitrarily good while the third fact tells us that we really have a pair of problems that belong together in a very strong sense. Let us establish these facts in order and start by 1. We have

$$b^T y \geq (Ax)^T y = x^T A^T y = x^T c = c^T x,$$

where the first inequality follows from $Ax \leq b$ and $y \geq 0$. We note that we have equality iff it is the case for any i that $y_i = 0$ or the i'th inequality is fulfilled by equality. Let us define such a solution and hence establish 2. Remember the simplex method and the condition for having an optimal solution. If the first n inequalities held with equality and the corresponding matrix was A_H then we had that $c^T A_H^{-1} e_i \geq 0$ for any i. We define $y_i = c^T A_H^{-1} e_i$ for $i \leq n$ and $y_i = 0$ for $i > n$. Thus for each i we either have $y_i = 0$ or that the inequality is satisfied with equality. Thus we only need to establish that $A^T y = c$. Since we have column vectors, $y_i = c^T A_H^{-1} e_i$ is more conveniently[1] written as $y_i = e_i^T (A_H^{-1})^T c$ giving $y = (A_H^{-1})^T c$ which is equivalent to $A^T y = c$ and we are done.

Finally let us establish 3. The dual is, by definition, given by

$$\min b^T y$$

$$\begin{aligned} A^T y &= c \\ y &\geq 0 \end{aligned}$$

[1]As the transpose of a number is the number itself.

Since we have only defined the dual for problems given by inequalities we rewrite the dual in the following form

$$\max -b^T y$$
$$\begin{pmatrix} A^T \\ -A^T \\ I \end{pmatrix} y \le \begin{pmatrix} c \\ -c \\ 0 \end{pmatrix}.$$

In the dual of the dual we have variables corresponding to the inequalities. If we call these α, β and γ, the dual of the dual is given by

$$\min c^T \alpha - c^T \beta$$
$$\begin{pmatrix} A & -A & -I \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix} = -b$$
$$\alpha, \beta, \gamma \ge 0.$$

The equality is written differently $A\alpha - A\beta - \gamma = -b$ and since this is the only occurrence of γ, except for $\gamma \ge 0$, we might drop γ and replace the equality by $A\alpha - A\beta \ge -b$. Finally setting $x = \beta - \alpha$ we get back the original problem. The condition that $\beta, \alpha \ge 0$ is simply dropped since it can always be made true by adding a large vector to both α and β and this does not change x.

One of the useful properties of the dual is that it turns out to be beneficial to solve the dual and the primal at the same time. We do not enter this discussion here.

Let us finally give just a hint of other algorithmic ideas. The shortcomings of the simplex algorithm is due to the fact that a polytope has a complicated boundary on which it is hard to find a good direction for improvement. In an attempt to avoid the boundary people have suggested algorithms that stay inside the polytope. The initial idea is due to Karmarkar[30] from 1984 and it has since created a large research area. To see how it would work suppose our problem is on the form

$$\min c^T x$$

$$\begin{aligned} Ax &= b \\ x &\ge 0 \end{aligned}$$

and that we are given an initial feasible point which is interior in that all x_i are strictly greater than 0. Now suppose we instead try to to solve

$$\min c^T x - \mu \sum_{i=1}^{n} \log x_i$$

$$Ax = b,$$

for some $\mu > 0$. Since $Ax = b$ just enables us to eliminate some of the variables this is any easy condition and one can find a local optima by Newton's method. The found minima satisfy $x_i > 0$ since the objective function tends to infinity if x_i tends to 0. Thus the found optimum belongs to K. Call the optimal point

$X(\mu)$. One can prove that as μ tends to 0, $X(\mu)$ will tend to the optimal point of the linear program, and that it will do so through the interior of K. It turns out that by adjusting parameters this idea can yield very efficient algorithms. They turn out to be provably polynomial time in the worst case and very efficient in practice.

Chapter 15

Matching in general graphs

General matching does not pose the requirements that the matching graph is bipartite. The set of vertices V represent a group of objects that we want to divide into pairs. The set of edges E represent a possible match between two objects. We cannot divide V into two distinct sets, since V represent only one type of object, consequently we can not use the algorithm for bipartite matching.

An example of general matching is a set of patrolling constables. These constables patrol in pairs, and every constable has preferences about his partner. Each constable is represented by a vertex v and there is an edge between vertices which give possible pairs. In order to utilize our resources in an efficient manner, we would like to have as many pairs out in the street as possible. In other words, maximize the size of a matching.

A natural algorithm to find the maximal size matching is to repeatedly find an augmenting path in the graph and update the matching accordingly. Remember that an augmenting path starts and ends at a free vertex (*i. e.* a vertex not in the matching) and every other edge in the path is contained in the current matching.

Then, what is *really* the difference between bipartite matching and general matching? In bipartite matching we could find an augmenting path with Depth First Search. The problem here is that we may reach vertices where we have to make a choice between several possible edges. (See Figure 15.1) An exhaustive search for all possible paths may then take as much time as $O(2^n)$.

The choice situation occurs when there is an cycle of odd length with every other edge in the current matching except for two adjacent edges which do not belong to the matching. This is the place we entered the cycle in our search for an augmenting path. The choice is essentially in which direction to go around this cycle. Such a cycle of odd length is called a *blossom* and the path that was the beginning of our search until we entered the blossom, is called the *stem*. (See Figure 15.2)

15.1 The idea for our algorithm

To find an augmenting path in a graph G we do a depth first search of G until we identify a *blossom* B. Then we create a new graph G' where B is replaced with a new vertex k. Continue the search in the same manner in G'. (See Figure

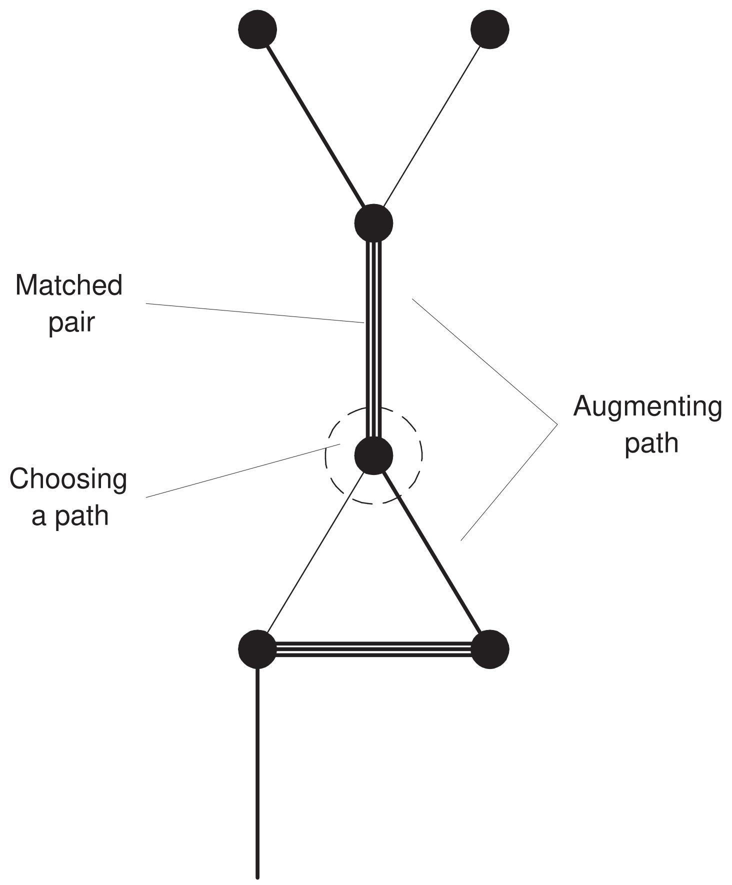

Figure 15.1: An augmenting path found by making a correct choice

15.2)

Lemma 15.1. *Given a graph G with a blossom B and a graph G' where B is substituted with a new node k.*

$$G \text{ has an augmenting path} \\ \Updownarrow \\ G' \text{ has an augmenting path.}$$

Proof. First we prove $\Uparrow$. Suppose G' has an augmenting path, P'. We can look at the edges in G corresponding to the edges of P' and we have the following cases:

1. If the path P' does not contain k, then the same edges give an augmenting path in G.
2. If the path P' ends in k, we get a corresponding path P in G, but it is not augmenting. Note that in this case, k was not in the matching of G', which imply that the blossom had not stem. This in turn implies that we can make the path P augmenting by walking around the blossom in the

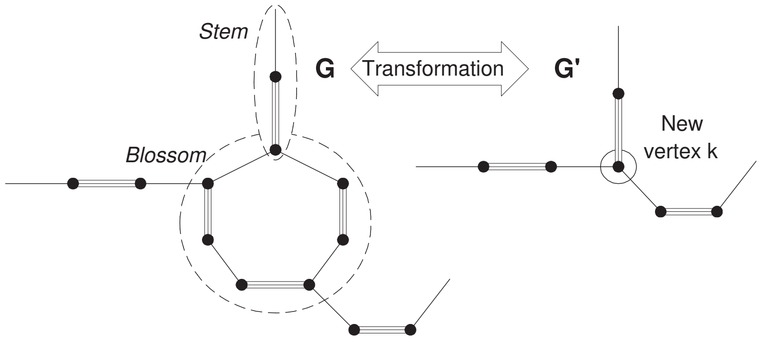

Figure 15.2: The general graph with a blossom

appropriate direction and end the path where the two unmatched edges connect.

3. If k is an internal vertex in P' then P' contains the final part of the stem where we entered the blossom in G. Consequently the augmenting path P' in G' corresponds to *two* paths in G, one that arrives to the blossom and one that ends where the stem is connected to the blossom. We can again connect these two into an augmenting path P by walking around the blossom in the appropriate direction.

Now let us prove $\Downarrow$. Suppose we have G and an augmenting path P in G. In G, we also have a blossom B possibly with a stem. Each vertex in the stem is called *odd* if the preceding edge was a matched edge and *even* if the preceding edge was unmatched. One end of P is not the endpoint of the stem and we begin to follow that end until we encounter either the blossom or the stem. We have the following cases:

1. If we encounter the blossom we change the augmenting path by continuing around the blossom in the appropriate direction and exit where the stem is connected. This is another augmenting path in G for which it is easy to see that it corresponds to an augmenting path in G', where the edges of the blossom are removed.

2. If we encounter the stem in an *odd* vertex, we can change P by replacing the remaining part of it by the part of the stem from the point P and the stem intersected to the starting point of the stem. Then we get an augmenting path that doesn't contain the blossom and we are finished.

3. If we instead encounter the stem in an *even* vertex we get into a more complicated situation than before. We proceed as follows:

   We replace the stem to the blossom with the part of the augmenting path P from the start starting point to where it encountered the stem. The rest of the stem (from the intersection point to the blossom) remains. stem. Now start all over again by traversing P (starting at its other endpoint). If we again encounter the stem in an even position we repeat the same procedure, *i. e.* , we replace part of the stem and start at the other endpoint of P (which is the one we had from the beginning). It might seem like this would lead to an infinite loop but the point is that the stem has changed meanwhile and hence we are making progress. To verify that this process eventually terminates we reason as follows. Since an augmenting path does not intersect itself for each iteration the intersection point moves at least on point closer to the blossom. Consequently sooner or later we either encounter the blossom or an odd vertex in the stem and we can apply the appropriate case as described above.

□

15.2 A more formal description of the algorithm

1. Start a *Depth first search* at a non matched vertex. Each visited vertex is labeled with an index i such that i is the length of the augmenting path so far at that vertex.

2. When an edge leads to an already visited vertex, we get two cases:

   (a) The vertex has and odd index i. This tells us that we have reached a Blossom B. The current graph G is transformed into graph G' by replacing B with a new node k and we restart from 1 with this new graph G'.

   (b) The vertex has an even index i. We exactly as when standard depth first search encounters and already visited vertex. In other words we keep exploring new edges from our current base.

3. When we complete an augmenting path in the current graph G we lift it to the original graph by using the algorithm implied by the reasoning in the proof of Lemma 15.1.

As an idea the algorithm is now completed, but alas, there are practical problems left concerning administration of data structures. For example, how do we represent the graphs and how do we create G'?

15.3 Practical problems with the algorithm

Let us answer the last question, how do we create G'? We consider the easiest approach, namely to write down the whole graph G' every time we encounter a blossom B in G.

1. Let k be the vertex replacing B.

2. Write to G' every edge $e(v, w) \in G$ iff $v \notin B \wedge w \notin B$. That is, every edge in G that is not part of the blossom.

3. Write to G' every edge $e(k, v) \in G$ iff $\exists w \in B \wedge e(w, v) \in G$. That is, every edge in G connecting to the blossom B, is reconnected to the new vertex k in G'.

Then, what is the time complexity for this naive implementation of our algorithm? The time it takes to create G' is $O(n^2)$. If we make consecutive $G \to G'$ transformations we get series of graphs: $G, G', G'' \dots G^{(l)}$. Finally we find an augmenting path in $G^{(l)}$, where $l \leq n/2$ since $G^{(i)}$ has at least two vertices less than $G^{(i-1)}$.

As mentioned earlier, $G^{(i)}$ is created from $G^{(i-1)}$ in time $O(n^2)$. Going backwards and lifting the augmenting path from $G^{(i)}$ to $G^{(i-1)}$ is done in time $O(n)$. The time for finding an augmenting path is then $O(n^3)$ and the total cost is $O(n^4)$ since we have $\leq n$ augmenting paths to find.

The largest cost appears when we do the $G^{(i)} \to G^{(i+1)}$ operation. The key to a better implementation of the algorithm is thus to find an improvement of the $G \to G'$ operation. This can be done by storing, in an efficient way, only the difference between G and G'. Being careful this can be done in time $size(Blossom) \cdot O(n)$ and this improves the overall cost to $O(n^3)$. We omit the details but take the liberty of stating the result as a theorem.

Theorem 15.2. *We can find a maximal size matching in general graphs with n nodes in time $O(n^3)$.*

The general idea for this algorithm was originally proposed by Edmonds [17]. The fastest known algorithm of today is by Micali and Vazirani [35] and runs in time $O(\sqrt{n} \cdot m)$.

Chapter 16

The Traveling Salesman Problem

The traveling salesman problem (TSP) is given by n cities where the distance between cities i and j is d_{ij}. We want to find the shortest tour that visits every city exactly once. One key property on instances of TSP is whether the distances fulfill the triangle inequality, i.e. $d_{ij} \leq d_{ik} + d_{kj}$ for any i, j and k. In every day terms it says that when going from city i to city j it is never cheaper to go through k than to go directly. This inequality is natural and true in most applications and we assume it most of the time.

An even more stringent form is that we have actual points x_i in a Euclidean space and that d_{ij} is the Euclidean distance between x_i and x_j. It is well known that TSP also in this special case is NP-complete and hence we cannot expect a fast algorithm that always finds an optimal solution. In this chapter we mostly discuss heuristic approaches to TSP and we rely heavily on material from the detailed survey [27] by Johnsson and McGeoch. However, [27] is over 100 pages and we only scratch the surface of the wealth of material presented there.

16.0.1 Finding optimal solutions

An optimal solution can be found in time around 2^n using dynamic programming and for the Euclidean case there exists an algorithm that has the time complexity $2^{c\sqrt{n}logn}$. Both these algorithms are, however, mostly of theoretical interest. When it comes to finding optimal solutions in practice other methods are used. The most successful method has been based on linear programming. Let us, very briefly discuss this line of attack.

For each edge (i, j) in the linear program we have a variable[1] x_{ij}. It should take the value 1 if the edge is part of the tour and 0 otherwise. We want to minimize $\sum_{ij} x_{ij} d_{ij}$ subject to the constraint that the x_{ij}'s describe a tour. We have a linear objective function and if the constraint of being a tour had been a linear constraint describable in a simple way we could have solved the problem exactly efficiently since linear programming can be done in polynomial time. As TSP is NP-complete this is too much to hope for and the approach is to use

[1]Since we have undirected edges there is no difference between the edge (i, j) and (j, i) and thus we assume that $x_{ij} = x_{ji}$.

an iterative method adding more and more relevant linear constraints on the variables x_{ij}. Consider the linear constraints

$$0 \leq x_{ij} \leq 1$$
$$\sum_j x_{ij} = 2, \text{ for } i = 1, 2 \dots n.$$

These are clearly true for any x_{ij}'s encoding a correct tour. On the other hand there are values of the variables that satisfy these conditions, but do not correspond to tours. There are basically two different reasons for variable assignments not corresponding to tours. The first is that the x_{ij} take values that are not either 0 or 1 and in particular taking $x_{ij} = \frac{2}{n}$ for all i and j gives a feasible solution. What turns out to be a more serious problem (we do not here enter the discussion why) is that even in the case when x_{ij} do only take the values 0 and 1 they might still not code a tour. The problem is that the edges with $x_{ij} = 1$ might not give one complete tour but many short disjoint tours.

Suppose that we, although we know that we do not have all the necessary constraints, solve the linear program and obtain an optimal solution. This solution probably does not code a tour. In this case we add additional linear constraints violated by the given optimal solution but which are true for all solutions that correspond to tours. Examples of such constraints are

$$\sum_{ij,\ i \in S\ j \notin S} x_{ij} \geq 2,$$

for any nonempty set S which is not all nodes. This is called a subtour elimination constraint and makes sure that we do not have closed subtour on the elements in S. Clearly any such constraint is valid for a correct coding of a tour. These constraints are exponentially many and it is not feasible to include them all. However, when we get an optimal solution which is a subtour on a certain S we can include this particular constraint. Running the program again gives a new optimal solution (with larger optimum) and if this solution does not correspond to a feasible solution either we add more constraints and iterate. Added constraints are called "cuts" and there are many other types of cuts that can be added. For a more thorough discussion on this topic we refer to [3].

This approach gives the most efficient algorithms for solving TSP optimally and [3] reports solving a 13509 city problem given by actual cites in the US. The problem was solved on a network of computers and the computation time corresponded roughly to 10 years computing time on a 400 Mhz Digital Alpha Server 4100. Solving problems of size less than 100 is not a challenge for the best programs these days and even problems of size around 1000 are solved routinely with running times of the order of hours.

In this chapter we will discuss heuristics that get a reasonable solution quite quickly. With the above in mind this is interesting only in the case of at least 1000 cities and we should really think of n in the range 10^4 to 10^6. We should keep in mind that for the larger size we have problems storing all pairwise distances and algorithms running slower than $O(n^2)$ start to be impractical.

16.1 Theoretical results on approximation

As stated above TSP is NP-complete even in the case of 2-dimensional Euclidean distances. A lot of energy has been spent on trying to get efficient approximation algorithms with provably good properties. We say that an algorithm is a C-approximation algorithm if it, for each input x, finds a tour that is of length at most $C \cdot opt(x)$ where $opt(x)$ is the length of the optimal tour.

The following algorithm is a classical result in the area established already in 1976 by Christofides [10].

Theorem 16.1. *TSP with triangle-inequality can be approximated within 1.5 in polynomial time.*

Proof. We outline the algorithm. First construct the minimal spanning tree. The, consider the nodes which have odd degree in the minimal spanning tree and find a minimal weight perfect matching on those nodes where the weight of edge (i, j) is given by d_{ij}. Consider the graph of the minimal spanning tree together with this matching. This forms a graph with all degrees even. It is well known that such a graph has an Euler tour (i.e. a tour that uses each edge exactly once) and such a tour can be found in linear time. We construct this Euler tour and let the TSP tour be the sequence of vertices as they appear on this Euler tour. When a node appears more than once all occurrences after the first are ignored. This creates short-cuts, but since the triangle inequality holds the length of the constructed tour is at most the cost of the spanning tree plus the cost of the matching.

We claim that this cost is at most 1.5 the cost of the optimal tour. This claim is established by the fact that the cost of the spanning tree is at most the cost of the optimal tour and that the cost of matching is at most half the cost of the optimal tour. We leave the task to verify these claims to the reader.

The final fact needed to establish the theorem is that a minimal weight matching for general graphs can be found in polynomial time. We did not cover this problem in our section on matchings but it can in fact be found in time $O(n^3)$. □

Theorem 16.1 still represents the best provable factor when we have no structure apart from the triangle inequality. However for Euclidean problems we can do better.

Theorem 16.2. *For 2-dimensional TSP with Euclidean distances and any $\epsilon > 0$ we can find a $1 + \epsilon$ approximation in polynomial time.*

We do not prove this theorem but refer to the original paper by Arora [4]. The running time is $O(n(\log n)^{O(1/\epsilon)})$ and is based on dynamical programming. It has not yet been found to be practical for interesting values of ϵ.

On the negative side, one can establish that Theorem 16.2 cannot be extended to general distance functions that only satisfy the triangle inequality. We have the following result, established by Lars Engebretsen [18]. It applies to instances where distances are 1 and 2 and note that any such distance function automatically satisfies the triangle inequality.

Theorem 16.3. *Unless $P = NP$, for any $\epsilon > 0$ one cannot approximate TSP with distances 1 and 2 within $4709/4708 - \epsilon$ in polynomial time.*

Now we turn to the question of solving these problems in practice. There are two main approaches which can be combined. We have heuristics for constructing tours and local optimization techniques to improve already constructed tours.

16.2 Tour construction heuristics

There are very many heuristics for constructing tours. The paper [27] singles out four, since they consider other heuristics dominated by these. A heuristics dominates another if it finds better tours and runs faster. The heuristics are

1. **Nearest neighbor.** Start from any vertex and keep visiting the nearest vertex that has not been visited.

2. **Greedy.** Keep picking the shortest edge subject to the constraint that edges picked so far does not give a node of degree greater than two and does not create a cycle of length less than N.

3. **Clarke-Wright.** Start with a pseudo-tour in which an arbitrarily chosen vertex is the *hub* and the salesman returns to the hub after each visit to another city. For each pair of non-hub cities define the savings to be the amount by which the tour would be shortened if the salesman went directly between the two cities. We now proceed as in the greedy algorithm. We go through the non-hub city pairs in non-increasing order of savings, performing the bypass so as it does not create a cycle of non-hub cities or cause a non-hub city to become adjacent to more than two other non-hub cities. When we have a tour we terminate.

4. **Christofides.** Described in the proof of Theorem 16.1.

To evaluate these strategies one can prove worst case bounds. Above, we established the Christofides is never off by more than a factor 1.5. The other heuristics are much worse in this respect and the worst case performances are between $\Theta(\log n/\log\log n)$ and $\Theta(\log n)$. Another parameter is the running time of the heuristics and here Christofides has running time $O(n^3)$ while the others can be implemented in time $O(n^2)$ or possibly $O(n^2 \log n)$. Various tricks can be used to make the algorithm run quicker in parallel and in practice they run in time $o(n^2)$ (at least for Euclidean instances) and thus they do not even look at all possible edges.

We now turn to the experimental evaluation of these heuristics. We are interested in two parameters; running time and performance. Since we cannot exactly know the optimal value on each instance, it is not obvious how to measure performance. We use instead excess over something called the Held-Karp lower bound. This lower bound is obtained through linear programming techniques. It is never smaller than 2/3 of the optimum but it is usually much better and experimentally it seems to be within a single percent of the true value.

The experiments were done mostly on random instances. Either on random points in the plane with Euclidean distances or on a random distance matrix where each d_{ij} is picked uniformly and independently in $[0, 1]$. Since the latter distribution does not satisfy the triangle inequality it does not make sense to use Christofides in this case since it heavily relies on the triangle inequality while

	Average Excess over Held-Karp Lower Bound								
$n =$	10^2	$10^{2.5}$	10^3	$10^{3.5}$	10^4	$10^{4.5}$	10^5	$10^{5.5}$	10^6
	Random Euclidean Instances								
CHR	9.5	9.9	9.7	9.8	9.9	9.8	9.9	-	-
CW	9.2	10.7	11.3	11.8	11.9	12.0	12.1	12.1	12.2
GR	19.5	18.8	17.0	16.8	16.6	14.7	14.9	14.5	14.2
NN	25.6	26.2	26.0	25.5	24.3	24.0	23.6	23.4	23.3
	Random Distance Matrices								
GR	100	160	170	200	250	280	-	-	-
NN	130	180	240	300	360	410	-	-	-
CW	270	520	980	1800	3200	5620	-	-	-

Table 16.1: The quality of the ours for our 4 tour construction heuristics.

	Running time in Seconds 150 Mhz SGI Challenge								
$n =$	10^2	$10^{2.5}$	10^3	$10^{3.5}$	10^4	$10^{4.5}$	10^5	$10^{5.5}$	10^6
	Random Euclidean Instances								
CHR	0.03	0.12	0.53	3.57	41.9	801.9	23009	-	-
CW	0.00	0.03	0.11	0.35	1.4	6.5	31	173	670
GR	0.00	0.02	0.08	0.29	1.1	5.5	23	90	380
NN	0.00	0.01	0.03	0.09	0.3	1.2	6	20	120
	Random Distance Matrices								
GR	0.02	0.12	0.98	9.3	107	1400	-	-	-
NN	0.01	0.07	0.69	7.2	73	730	-	-	-
CW	0.03	0.24	2.23	22.0	236	2740	-	-	-

Table 16.2: The running times for our 4 tour construction heuristics.

the other heuristics do not. The quality of the tours are given in Table 16.1 and the running times are given in Table 16.2 Thus the tours are not very good but are, at least in the Euclidean case, acceptable. Running times are, however, very good and in the Euclidean case, even a million cities instance requires only minutes on the faster heuristics.

16.2.1 Local Optimization

A tour constructed by the above heuristics might be optimized using local optimization. Three main techniques are commonly used.

1. **2-opt.** For each pair of edges in the tour try to exchange them.

2. **3-opt.** Delete any set of three edges and try to reconnect the tour in all possible ways and see if any alternative produces a shorter tour.

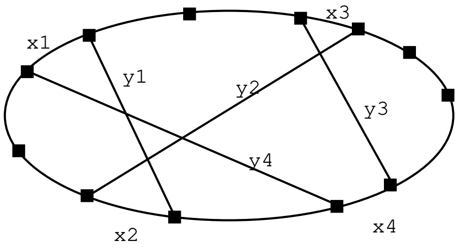

Figure 16.1: Exchanging x's to y's on a tour.

3. **Lin-Kernighan.** A rather complicated local search strategy described in next section.

Although both 2-opt and 3-opt are rather obvious to describe an efficient implementation requires more care. In particular one should not consider the $\Omega(n^3)$ possible choices for three edges in 3-opt explicitly since for $n = 10^6$ this would give 10^{18} cases which is impractical.

16.2.2 Lin-Kernighan

We make a slightly simplified but largely correct description of the algorithm proposed by Lin and Kernighan . We are given a suggested tour and we are trying to find a tour with lower cost. We delete some edges and replace them by some new ones. We do not limit the number of edges deleted but we restrict how they are found.

Consider the change as a switch from edges x_1 x_2..x_s to y_1 y_2 .. y_s where y_i has a vertex common with x_i and $x_i + 1$ and y_s has a vertex common with x_s and x_1 (See fig 16.1).

The decision of x_1 y_1 x_2 y_2 ... can be viewed as a tree. If the entire tree is searched we will find an optimal solution.

Lin-Kernighan wants to describe a part of the tree as follows:

1. The cost $\sum_{i=1}^{t} c(x_i) - c(y_i)$ should always be positive, *i. e.* changing from the x-edges to the y-edges decreases the total cost, but at intermediate stages this does not produce a tour.

2. For $t \geq 3$ choose x_t such that if y_t goes to the open endpoint of x_1 we get a tour.

3. Do total backtracking on level 1 and 2, otherwise choose the y_i with the lowest cost or an y_i that completes the tour.

If a better tour is found the basic tour is updated and we start all over again searching for an even better tour.

There are some details to take care of in the description. In particular is it true in Step 2 that there is a direction which satisfies this property and if so, how to determine this direction? The latter is clearly no problem since we can just try the two directions and see which one works. We leave the former problem to the reader.

For some details on how to get an efficient implementation we refer to [27].

	Average Excess over Held-Karp lower bound								
$n =$	10^2	$10^{2.5}$	10^3	$10^{3.5}$	10^4	$10^{4.5}$	10^5	$10^{5.5}$	10^6
	Random Euclidean Instances								
2-Opt	4.5	4.8	4.9	4.9	5.0	4.8	4.9	4.8	4.9
3-Opt	2.5	2.5	3.1	3.0	3.0	2.9	3.0	2.9	3.0
LK	1.5	1.7	2.0	1.9	2.0	1.9	2.0	1.9	2.0
	Random Distance Matrices								
2-Opt	34	51	70	87	125	150	-	-	-
3-Opt	10	20	33	46	63	80	-	-	-
LK	1.4	2.5	3.5	4.6	5.8	6.9	–	–	-

Table 16.3: Quality of tours for local optimization heuristics.

	Running Times in Seconds on a 150 Mhz SGI Challenge								
$n =$	10^2	$10^{2.5}$	10^3	$10^{3.5}$	10^4	$10^{4.5}$	10^5	$10^{5.5}$	10^6
	Random Euclidean Instances								
2-Opt	0.03	0.09	0.34	1.17	3.8	14.5	59	240	940
3-Opt	0.04	0.11	0.41	1.40	4.7	17.5	69	280	1080
LK	0.06	0.20	0.77	2.46	9.8	39.4	151	646	2650
	Random Distance Matrices								
2-Opt	0.02	0.13	1.02	9.4	108	1400	-	-	-
3-Opt	0.02	0.16	1.14	9.8	110	1410	-	-	-
LK	.05	0.35	1.9	13.6	139	1620	-	-	-

Table 16.4: Running times for local optimization heuristics.

16.2.3 Evaluation of local optimizations

There is again the choice of a theoretical and a practical evaluation. Since we are taking a rather practical approach we are really more interested in the latter but we should keep in mind that our overall perspective is theoretical. It turns out that if we do not assume the triangle inequality there are instances which are $1/4\sqrt{n}$ (2-opt) and $1/4n^{1/6}$ (3-opt) longer than optimal and still there are no local improvements. There are also examples of instances where the number of iterations of 2-opt before one reaches a locally optimal tour are as large as $2^{n/2}$. If we consider only Euclidean instances the situation is not as bad, but a locally optimal solution can still be a factor $\Omega(\log n/\log\log n)$ longer than the best tour.

Let us now turn to the experimental evaluation. We generate a starting tour by the greedy heuristic and then we perform local improvements. The excess over the Held-Karp bound is found in Table 16.3 and the running times are given in Table 16.4. The tables give, in our mind, very impressive numbers. Especially impressive is the combination of very good quality tours and the low running times. For the moment however, let us concentrate on a small detail.

Take the simplest optimization rule, 2-opt. On Euclidean instances, it runs in clearly subquadratic time. Implemented naively it would use $\Omega(n^2)$ steps per iteration. This would lead to $\Omega(n^3)$ overall running times since the number of iterations turn out to be $\Theta(n)$ in practice. Let us briefly discuss some tricks used in obtaining these very fast running times.

For a detailed description it is convenient to think of the tour as directed, so let us take some arbitrary direction of a given tour. A switch in 2-opt is given by two edges (t_1, t_2) and (t_3, t_4) that are switched to (t_1, t_3) and (t_2, t_4). Note that if we connect t_1 to t_4 we create two disjoint cycles (remember that we think of the tour as directed). For the switch to be profitable we need

$$d_{t_1 t_3} + d_{t_2 t_4} < d_{t_1 t_2} + d_{t_3 t_4}.$$

This implies that we need that $d_{t_1 t_3} < d_{t_1 t_2}$ or $d_{t_2 t_4} < d_{t_3 t_4}$ or possibly both. The first condition would imply that one of the neighbors of t_1 is changed to a closer neighbor and the second gives the same conclusion for t_4. Since the two tuples (t_1, t_2, t_3, t_4) and (t_3, t_4, t_1, t_3) give the same switch we can go through the edges of the present tour and try to change a given edge to an edge that is shorter. Since most vertices are connected to one of their closest neighbors this limits the number of possible choices severely. This observation only speeds up the search without making oversimplifications.

One simplification connected with the above operation is to, for each i, store the k closest cities to city i and only use these for updates. Storing such a neighborlist will supply most candidates and be very efficient. It turns out that already $k = 20$ is sufficient in many circumstances and using a k above 80 tends to add very little. Another shortcut is to use a bit b_i to tell whether at all it is useful to try to find a new neighbor for city i. These bits are all true initially but if we try to find a new neighbor of i and fail we make b_i false. It is not made true again until one of the neighbors of i in the tour change their status. With this convention we might miss some fruitful moves but the speedup is significant.

It turns out that with all these tricks, the main bottleneck is to update the tour. If we simply store the tour in an array, each 2-opt move can cost time $\Omega(n)$ leading to a quadratic running time. The method proposed is to keep a two level tree of degree $\sqrt{n}$. The vertices of the tour are given by the leaves of the tree read in a certain order. The children of each node are sorted from left to right. To get the correct order for the TSP tour you should process the middle level nodes from left to right but for each middle level node there is a bit that tells you whether you should read its children from left to right or the other way around. It is easy to see that you can update this structure at a cost of $O(\sqrt{n})$ for each local optimization step.

Clearly to get the good running times of 3-opt and Lin-Kernighan a number of similar tricks have to be used. We refer to [27] for details.

If we have more computer time to spend and are interested in obtaining even better solutions there are a number of general approaches one can try. Let us mention a few famous approaches and define them with a few words.

1. **Tabu search.** Suppose we work with a local search algorithm like 2-opt. The problem is that we get stuck in a local optima and we cannot continue. The idea with tabu search is to escape the local minima by allowing the

algorithm to go to worse solutions. To avoid cycling, when doing such a move one somehow creates some tabus (i.e. disallowing some moves in the future). This can be done in a number of different ways and we again refer to [27] for some details related to TSP.

2. **Simulated annealing.** Simulated annealing is a randomized local search strategy where we allow the search to go to worse solutions. It is modeled after physical systems and the system is guided by a temperature. The probability of going to a worse solution is smaller than that of going to a better solution and the ratio of the probabilities of going to various solutions is guided by the temperature and their total costs. For high temperatures it is quite likely that the algorithm chooses to go to worse solutions while when the temperature reaches 0 the system never goes to a worse solution. The idea now is to start at a high temperature and slowly cool the system and hope that the system reaches a good solution.

3. **Genetic algorithms.** This set of algorithms are inspired by evolution. We have a population of individuals which in this case are possible tours. These tours can then combine through mating and the most viable (*i. e.* shortest) offspring is more likely to survive. This is simulated for a number of generations. The best tour ever produced is the answer used. To make this idea competitive for TSP we have to add a non-evolutionary property by allowing an individual to improve itself. This is achieved by running a local optimization of the individuals after mating.

4. **Neural networks**. This is also a loosely defined set of algorithms where one makes an artificial nervous system and tries to train this nervous system to perform well. This has not been very successful so far for TSP and one reason could be that these algorithms are really best for tasks for which humans still are superior to computers. Driving a car or recognizing a face are typical such tasks while combinatorial optimization is not.

We are really looking for an algorithm that produces better tours than Lin-Kernighan but to make a fair comparison we have to remember that none of the above approaches can compete with Lin-Kernighan in terms of speed. One way to make the comparison more fair is to allow Lin-Kernighan to start with many different starting tours and perform local improvements until it gets to a local optimum. We would then simply choose the best among a number of runs. It turns out that there are really two methods that can compete with Lin-Kernighan. One is simulated annealing for some range of the parameters. This is, however, in the range where we spend a fair amount of time to find really good solutions for n in the range from 1000 to 10000 and the algorithm gets impractical for larger n. The more serious competitor is a genetic algorithm called Iterated Lin-Kernighan (ILK) that produces offspring from a single parent by making a suitable change and then runs Lin-Kernighan as a local optimization routine. This is rather similar to applying Lin-Kernighan to many different starting tours but experience seems to indicate that it is superior. Our final set of experimental results comes for comparing Lin-Kernighan (with independent starting points) to ILK. We keep a parameter that is the number of independent runs and the number of iterations respectively. We make this a function of n and run our algorithms on random Euclidean instances. The quality of the

Average Excess over Held-Karp lower bound							
Indep. iterations	10^2	$10^{2.5}$	10^3	$10^{3.5}$	10^4	$10^{4.5}$	10^5
1	1.52	1.68	2.01	1.89	1.96	1.91	1.95
$n/10$	.99	1.10	1.41	1.62	1.71	-	-
$n/10^{.5}$	.92	1.00	1.35	1.59	1.68	-	-
n	.91	.93	1.29	1.57	1.65	-	-
ILK iterations							
$n/10$	1.06	1.08	1.25	1.21	1.26	1.25	1.314
$n/10^{.5}$	.96	.90	.99	1.01	1.04	1.04	1.08
n	.92	.79	.91	.88	.89	.91	-

Table 16.5: The quality of solutions of repeated and iterated Lin-Kernighan.

Average Excess over Held-Karp lower bound							
Indep. iterations	10^2	$10^{2.5}$	10^3	$10^{3.5}$	10^4	$10^{4.5}$	10^5
1	.06	.2	.8	3	10	40	150
$n/10$	.42	4.7	48.1	554	7250	-	-
$n/10^{.5}$	1.31	14.5	151.3	1750	22900	-	-
n	4.07	45.6	478.1	5540	72400	-	-
ILK iterations							
$n/10$	.14	.9	5.1	27	189	1330	10200
$n/10^{.5}$	.34	2.4	13.6	76	524	3810	30700
n	.96	6.5	39.7	219	1570	11500	-

Table 16.6: The running times of repeated and iterated Lin-Kernighan.

tours and the running times are found in Table 16.5 and Table 16.6 respectively. We suspect that the better running times for ILK are due to the fact that the Lin-Kernighan procedure starts from a better tour each time.

16.3 Conclusions

It is clear that by a combination of good technique, insightful ideas and good programming TSP can be solved very efficiently and accurately. It can be solved exactly for problems of sizes in the range of single thousands and within a few percent for sizes up to a million. One would almost be tempted to conclude that TSP, at least for Euclidean instances is easy in practice. Clever implementations of Lin-Kernighan or even 3-opt will take you a very long way and unless you are looking for optimal or very close to optimal solutions this is probably all you need.

We also would like to point to the tremendous improvements in the performances the best algorithms over the last decade. A natural reason is of course the improvement in hardware, but our opinion is that improvements in software and algorithms have helped at least as much. In our view there are also many existing ideas that have not yet been pushed to their limits and probably also new ideas to be discovered. Thus we should expect further improvements in algorithms and software. This together with the constant improvement in hardware will probably make it possible to solve TSP optimally on much larger instances in a not too distant future. A hope that Euclidean instances with 100000 cities should be solvable optimally within 10 years does not seem overly optimistic, but it is an amateur's guess.

This chapter also sheds some light on the practical shortcomings of the theory of NP-completeness. That a problem is NP-complete implies that there is no algorithm that runs in polynomial time and always outputs the optimal solution. It does not exclude the possibility that there is a linear time algorithm that solves 99% of the instances we are interested in!

Chapter 17

Planarity Testing of Graphs

17.1 Introduction

We are interested in the following problem: given an undirected graph G, determine if G is planar ("can be drawn on a paper without having any edges cross").

The algorithm developed in this chapter attempts to embed G in the plane. If it succeeds it outputs the embedding (at this point we are deliberately vague about exactly what this means) and otherwise it outputs "no." This is done by embedding one path at a time where the endpoints on the path are vertices that have already been embedded. Which path to embed, and how to embed it, is chosen in a greedy fashion. The difficulty is to show that the greedy choice used always finds an embedding, assuming one exists. The complexity of the algorithm is $O(|V|^2)$.

Hopcroft and Tarjan [24] present an algorithm that solves the problem in time $O(|V|)$ based on a previous algorithm [5, 21]. The algorithm presented here uses the same basic ideas but is less complicated.

17.2 Some fragments of graph theory

Let $G = (V, E)$ be a planar graph that has been embedded in the plane. The graph divides the plane into regions called *facets*. We let F be the set of facets. Note that the infinite region surrounding the graph is also a facet. See Figure 17.1.

The following relation, known as **Euler's formula**, holds for any planar graph $G = (V, E)$ (except for $|V| = 0$) where c is the number of connected components in G:

$$|V| + |F| - |E| = 1 + c.$$

This can easily be proven by induction.

Lemma 17.1. *If $G = (V, E)$ is a planar graph, then $|E| \leq 3|V| - 6$.*

Proof. We use a simple counting argument. Assume that G has been embedded in the plane and that F is the set of facets. An edge $e \in E$ *touches* a facet $\mathsf{F} \in F$ if e is in F's cycle (*i. e.* the cycle that forms the boundary of F). Define H

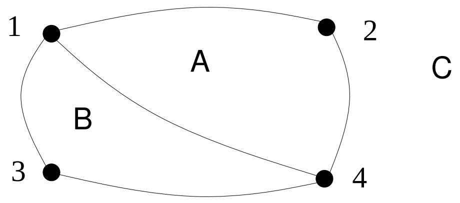

Figure 17.1: A graph embedded in the plane. $V = \{1, 2, 3, 4\}$, $E = \Big\{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 4\}, \{3, 4\}\Big\}$, and $F = \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}$.

as $\{\, (e, \mathsf{F}) \mid e \text{ touches } \mathsf{F} \,\}$ and we want to establish bounds on $|H|$. Since every edge touches at most two facets we have $|H| \leq 2|E|$, and since every facet is touched by at least three edges we have $3|F| \leq |H|$ which implies

$$3|F| \leq 2|E|.$$

Combining this with Euler's formula (and observing that G has at least one connected component) proves the lemma. □

From Euler's formula and the above lemma it follows that a connected planar graph has $|E| = \Theta(|V|)$, and for any embedding $|F| = O(|V|)$.

An undirected graph is biconnected if it is connected and at least two vertices must be removed to make it disconnected. Using depth-first search one can find all biconnected components of a graph (this induces a partition of E) in time $O(|V| + |E|)$. The following lemma explains the importance of biconnected components in planarity testing.

Lemma 17.2. *Let G be an undirected graph. Then G is planar if and only if each biconnected component is planar.*

This is easy to prove simply by observing that each biconnected component can be embedded in the plane separately.

17.3 A high-level description of the algorithm

The main part of the algorithm is the algorithm Embed which attempts to embed a biconnected graph in the plane.

Algorithm Embed

1. If $|E| \geq 3|V| - 6$ then output "no" and halt.
2. Find a (simple) cycle in G and embed it in the plane.
3. For each connected component C_j of un-embedded edges, let V_j be the set of embedded vertices that are incident on edges in C_j, and let F_j be the set of possible facets for C_j (a facet F is possible for C_j if all vertices in V_j are on the bounding cycle of F).
4. If $|F_j| = 0$ for any j, then output "no" and halt.

5. If $|F_j| = 1$ for some C_j, then take any path in C_j between two vertices in V_j and embed it in the single facet in F_j. If there are components left then go to 3, otherwise output the embedding.

6. If all components have $|F_j| \geq 2$ then pick a component C_i and a path from it and embed the path in some facet in F_i. If there are components left then go to 3, otherwise output the embedding.

The assumption that Embed deals with a biconnected graph implies that $|V_j| \geq 2$ for each component C_j in step 3.

There are several issues remaining to take care of. First of all we need to specify how to choose a component and a facet in step 6. Then we need to prove that the algorithm does not make bad choices in this step, that is, it finds an embedding if one exists. Finally, we need to show that the complexity of the algorithm is $O(n^2)$. We end this section by answering the first of these questions: in step 6 the component C_i, the path, and the facet in F_i can be chosen arbitrarily. In the following sections we argue that if an embedding exists then the algorithm finds it, and show that by somewhat careful bookkeeping the algorithm has complexity $O(n^2)$.

Algorithm Embed assumes that the graph is biconnected. To handle the case when G is not biconnected we introduce some pre and post processing.

Algorithm Main

1. If $|E| > 3|V| - 6$ then output "no" and halt.

2. Divide G into biconnected components $G_j = (V_j, E_j)$.

3. Run algorithm Embed on each component G_j. If Embed returns "no" for any G_j then output "no" and halt.

4. Combine the embeddings returned by Embed to a single embedding and output it.

17.4 Step 6 of Embed is sound

In this section we show that the choices in step 6 of the algorithm Embed can be made arbitrarily. The following is easy to check.

Lemma 17.3. *If we run the algorithm on a biconnected graph G, then, at any stage, the part that has been embedded is a biconnected graph.*

Now we get to the main lemma.

Lemma 17.4. *Let G be a biconnected graph which is given to the algorithm Embed. Assume that the algorithm reaches step 6 at time T and that we choose to embed a path from a component C_i that can be embedded in several facets. If G can be embedded by putting C_i in facet R and facet B is another possible facet in F_i, then G can be embedded by putting C_i in B.*

Proof. Both R and B are bounded by simple cycles since the embedded part of G is biconnected. We divide the vertices on these cycles into three groups: red, blue, and purple. The red vertices are those that occur in R's cycle but not in

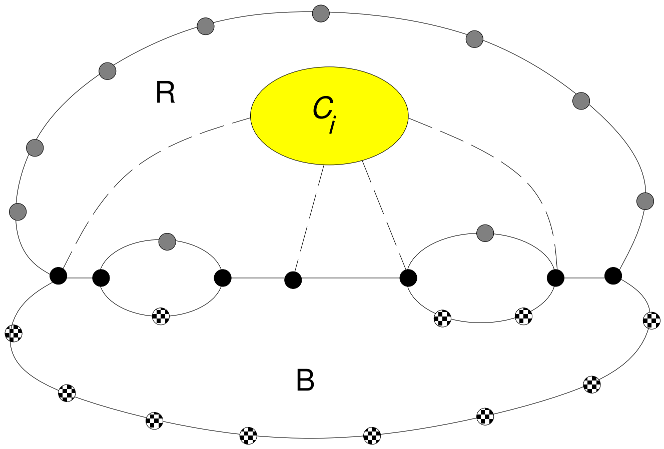

Figure 17.2: ● = red, ⊗ = blue, ● = purple. The component C_i has both R and B as possible facets. (Only the vertices and edges belonging to R and B are shown.)

B's, the blue those that occur in B's cycle but not in R's, and the purple vertices are those that appear in both cycles. See Figure 17.2 for an example.

Since C_i has both R and B as possible facets, V_i must be a subset of the purple vertices. The components C_j that are un-embedded at time T and for which V_j consists only of purple vertices are called *purple components*.

Assume that we are given an embedding $\mathcal{E}$ which is consistent with the partial embedding that the algorithm has produced at time T and which embeds C_i in R. Consider the components that are un-embedded at time T and which $\mathcal{E}$ puts in either R or B. The purple ones are called *active* and the others are called *inactive*. We claim that we can get another embedding, where C_i is in B, by "reflecting" all active components, that is, flipping them from R to B and vice versa. Clearly this operation cannot cause a conflict between two active components or two inactive components. The only possibility is a conflict between an active component and an inactive component. We show that no such conflict occurs in R, and by symmetry not in B.

Consider R's cycle. It is divided into *segments* in the following way. A purple segment is a maximal path of purple vertices. A red segment is a path of length ≥ 3 that has purple endpoints but all internal vertices are red (note that the endpoints of a segment are also endpoints of other segments).

Let C_l be an arbitrary active component in B and C_k an arbitrary inactive component in R. Assume that flipping C_l causes a conflict with C_k. Since C_k is inactive, V_k contains at least one red vertex that belongs to some segment S. If $V_k \subseteq S$, then there is no conflict with C_l, since C_l does not attach to any internal vertex of S. Thus, if there is a conflict, there must be some vertex (red or purple) in V_k that is not in S. However, in this case it is easy to see that at time T R is the only possible facet for C_k, and this contradicts the assumption that we have reached step 6. $\square$

The following theorem follows by straight-forward induction from Lemma 17.4.

Theorem 17.5. *Algorithm Main finds a planar embedding of G if one exists, and output "no" otherwise.*

17.5 Analysis of the algorithm

First it is necessary to specify how a (partial) embedding is represented. For each embedded vertex we have a list of its embedded neighbors and facets ordered clockwise. We call this a neighbor–facet list.

We also need some data structures for internal use in Embed. Rather than recomputing the components C_j and the sets V_j and F_j we maintain a list of all un-embedded components, and for each component C_j a list F_j of its possible facets. For each facet F we maintain a list L_{F} of components that have F as a possible facet. The sets V_j are not necessary to maintain.

Analysis of Main

Step 1 can certainly be done in time $O(|V| + |E|) = O(|V|^2)$ and with just about any reasonable representation of the input it can be done in time $O(|V|)$.

Step 2 can be done by depth-first search and hence in time $O(|V| + |E|)$ which is $O(|V|)$ by step 1.

A call to Embed costs $O(|E_j|^2)$ which is shown below. Since $\sum_j |E_j| = |E| = O(|V|)$, the cost of step 3 is $O(|V|^2)$.

Finally we claim without proof that step 4 can be done in time $O(|V|)$.

Analysis of Embed

The input is a biconnected graph $G = (V, E)$. We need to show that embed can takes time $O(|E|^2)$.

Step 1 is easily done in $O(|V| + |E|)$, which is $O(|E|)$ since G is connected. As before, this can in fact be done in time $O(|V|)$.

Let $n = |V|$. After step 1 we have $|E| = O(n)$.

To find a cycle in step 2 only requires depth-first search which takes time $O(n)$. This is also sufficient time to construct the neighbor–facet lists for the vertices in the cycle, since each such list contains only two vertices and two facets. Again using depth-first search, we find all the connected components C_j and add them to a list *Comps*. For each C_j on *Comps*, let F_j be the list of the two facets A and B (inside and outside the cycle). Let L_{A} and L_{B} both be the list of all components. There are $O(n)$ lists of size $O(1)$ and $O(1)$ lists of size $O(n)$. The lists are easy to construct in time $O(n)$.

The steps 3 through 6 form a loop which is executed at most $|E| = O(n)$ times, since each iteration embeds at least one edge. We want to show that every step can be done in time $O(n)$.

Step 3 is actually unnecessary. We already have the lists that are needed, except for V_j, and these sets are not needed explicitly.

Step 4 only involves checking if F_j is empty for any j, which is clearly doable in time $O(n)$ since there are $O(n)$ such lists. The same thing is true for checking if $|F_j| = 1$ for any j in step 5. If the algorithm checks if lists are empty or only contain one element after every update and maintains a list of components for which $|F_j| = 1$, then these steps only cost $O(1)$.

The only thing we need to worry about in step 5 and step 6 is the cost of finding the path in C_i and performing the necessary updates of our data structures. The path in C_i can be found by depth-first search in time linear in the size of C_j, which is certainly $O(n)$.

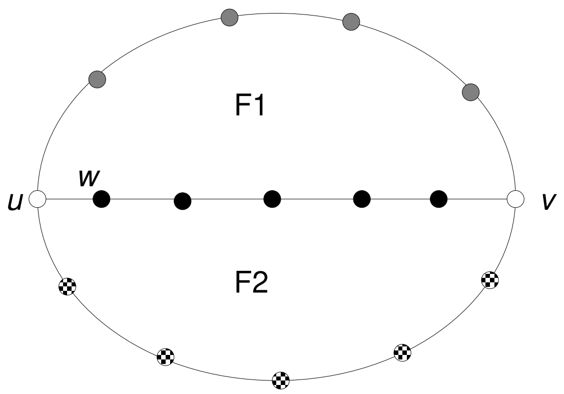

Figure 17.3: The facet F is split into F1 and F2 by embedding a new path. ● = type I, ⊗ = type II, ● = type III.

Embedding the path in a facet F splits it in two: F1 and F2 (see figure 17.3). Let u and v be the endpoints of the path, and let K be the cycle that is the boundary of the facet F, and let P be the path that is to be embedded in F.

Let w be u's neighbor on the new path. Update u's neighbor–facet list by replacing F by the sublist $\langle \mathsf{F1}, w, \mathsf{F2} \rangle$. Do similarly for v. Then, starting at u walk along K clockwise until v is reached. This can be done using the neighbor–facet lists. As we do this we replace each occurrence of F in an intermediate vertex's list by F1. Also, mark each internal vertex on this u, v-path as "type I." Then walk from v to u clockwise along K replacing F by F2 and mark internal vertices as "type II." The cost of this traversal is $O(deg(x))$ for each vertex x on K. Since there are $O(n)$ edges in the graph the combined cost for the traversal is $O(n)$.

Then mark all the internal vertices on P as "type III"[1] and construct their neighbor–facet lists (which is easy since they have two neighbors and F1 and F2 as their only facets). This clearly costs $O(n)$.

Finally, *Comps* must be updated, the lists F_j must be modified for all components C_j that have F as a possible facet, and the lists L_{F1} and L_{F2} must be constructed. This is accomplished as follows. Make two empty lists L_{F1} and L_{F2}. Remove the component C_i, from which P was chosen, from *Comps*. C_i has been split into components C_{i_k} by the embedding of P. These components can be found by a depth-first search in C_i. Each new component is added to *Comps*. Note that each C_{i_k} has at least one type III vertex, so the only facets that could be possible for a new component is F1 or F2. Do a depth-first search of each component C_{i_k} to find V_{i_k}. By looking at the type labels on the vertices in V_{i_k} the lists F_{i_k} can be constructed as follows.

- If V_{i_k} contains both a type I vertex and a type II vertex, then F_{i_k} is set to an empty list (in this case we could halt the algorithm since it is impossible to embed C_{i_k}).
- If V_{i_k} contains a type I vertex but no type II vertex, then set F_{i_k} to $\langle \mathsf{F1} \rangle$ and add C_{i_k} to L_{F1}.

[1]The type III marking is not necessary, but it helps in the discussion.

- If V_{i_k} contains a type II vertex but no type I vertex, then set F_{i_k} to $\langle \mathsf{F2} \rangle$ and add C_{i_k} to L_{F2}
- If V_{i_k} contains neither a type I vertex nor a type II vertex, then set F_{i_k} to $\langle \mathsf{F1}, \mathsf{F2} \rangle$ and add C_{i_k} to L_{F1} and L_{F2}.

It remains to handle all the "old" components C_j that had F as a possible facet. Note that C_j cannot attach to a type III vertex. For each component C_j on L_{F}, do a depth-first search to check if V_j contains any type I or type II vertex.

- If V_j contains neither type, then remove F from F_j and add $\mathsf{F1}$ and $\mathsf{F2}$ (*i. e.* , C_j attaches only to u and v), and add C_j to L_{F1} and L_{F2}.
- If V_j contains both type I and type II, then remove F from F_j.
- If V_j contains type I but not type II, then remove F from F_j, add $\mathsf{F1}$ to F_j, and add C_j to L_{F1}.
- If V_j contains type II but not type I, then remove F from F_j, add $\mathsf{F2}$ to F_j, and add C_j to L_{F2}.

Both for old and new components are searched in linear time and no edge appears in more than one component. This means that there are $O(n)$ insertions and deletions. We do not go into such detail as to show how the data structures should be implemented to allow each operation in $O(1)$ time. This is however possible using various cross-references between items in lists.

To summarize, the initial step of Embed costs $O(n)$, and each of the $O(n)$ iteration costs $O(n)$ showing that the complexity of Embed is $O(n^2)$.

Chapter 18

Sorting

18.1 Introduction

When studying algorithms on sorting, searching and selection there are at least two interesting characteristics. The first is whether the algorithm is comparison based or whether it uses general operations, like arithmetic and indirect addressing. The other interesting property is whether the algorithm is efficient in the worst case or only in the average case. In this chapter we consider sorting from these two perspectives.

18.2 Probabilistic algorithms

We here consider probabilistic algorithms which always yield the correct answer, while the running time is a random variable. On main parameter of study is the expected running time.

We have two sources of randomness.

1. Random input.

2. The algorithm makes its own random choices.

Consider quicksort . Remember that the basic step of quicksort is to choose x from the list and then perform the following operations.

1. $S_1\{y|y < x\}$

2. $S_2\{y|y \geq x\}$

3. Return $Qs(S_1), x, Qs(S_2)$

If x always is the first element in the list, the algorithm is of type 1. If x is chosen randomly from the list, the algorithm is of type 2.

In general type 2 is better than type 1 since it is easier to control the internal random choices than to make sure that the input is random. Comparison algorithms of type 1 can, however, be converted to type 2 by making a random permutation of the inputs before applying the type 1 algorithm.

18.3 Computational models considered

We consider three different computational models.

1. Comparison based algorithms that are efficient in the worst case.

2. Comparison based algorithms that are efficient in the average case.

3. General algorithms efficient for random input.

We could also discuss general algorithms for worst case inputs but this turns out not to be very fruitful in that it is hard to use general operations in an efficient way for sorting if the input has no special structure.

It is well known that the time complexity for sorting algorithms is $O(n \log n)$. Both mergesort and heapsort are of this type, and if we only count comparisons they give the constant 1 (provided that the base of the logarithm is 2).

The bound $n \log n$ is tight, since a sorting algorithm that uses T comparisons can give at most 2^T answers. By the answer we mean here how to permute the input to make the numbers appear in sorted order. Since there are $n!$ permutations we need $2^T \geq n!$ and using Stirling's formula, this gives $T \geq n \log_2 n - n \log_2 e + o(n)$.

For model 2, essentially the same lower bound applies, and thus randomness does not help sorting, at least not for comparison based algorithms.

Let us now turn to the third model i.e general algorithms on random inputs. We assume that we are given n inputs drawn independently from the same probability distribution. We assume that the probability distribution is known and we claim that in this case which distribution we have is of small significance.

18.4 The role of the Distribution

In the section following this one, we assume the distribution of the input to be uniform on $[0, 1]$. If the input is from another distribution we transform it. As an example assume that the inputs are chosen from $N(0, 1)$ (normally distributed with mean 0 and standard deviation 1). We then replace x by $P(x) = Pr[y < x]$ where y is $N(0, 1)$. This transforms the distribution of the data to be uniform on $[0, 1]$ while preserving the order of the elements.

This transformation works for any distribution such that $Pr[y < x]$ can be calculated efficiently. It is not hard to see that in practice it is sufficient to have an approximation of this number.

18.5 Time complexity of bucketsort

Bucketsort is a deterministic algorithm that works well for random inputs. It is a general algorithm in that it uses arithmetic and indirect addressing. As discussed above we assume that the inputs $x_1, x_2, \ldots x_n$ are uniformly distributed on $[0, 1]$.

Bucketsort has n buckets $(B_j)_{j=1}^n$ and if we let $\lceil x \rceil$ be the smallest integer not smaller than x we have the following description.

1. For $i = 1, 2, \ldots n$ put x_i into B_j where $j = \lceil x_i * n \rceil$.

2. For $j = 1, 2, \ldots n$ sort B_j.
3. Concatenate the sorted buckets.

We first claim that the algorithm is obviously correct and we are interested in estimating its running time.

Theorem 18.1. *When given n inputs drawn uniformly and independently from $[0, 1]$, bucketsort runs in expected time $O(n)$.*

Proof. The only operation that might take more than $O(n)$ time is sorting the lists B_j. To analyze this part, assume that B_j contains N_j elements. Even if we use a sorting algorithm running in quadratic time the total running time for sorting the buckets is

$$O(n) + c \sum_{j=1}^{n} N_j^2. \tag{18.1}$$

We use a combinatorial argument to estimate this number. N_j^2 is the number of pairs (i, k) (unordered and taken with replacement) such that both x_i and x_k are put into B_j. This means that the sum in (18.1) is the total number of pairs (i, k) such that x_i and x_k are put into the same bucket. We have n pairs (i, i) that always land in the same bucket and $n^2 - n$ pairs (i, j) where $i \neq j$ that land in the same bucket with probability $1/n$. This means that:

$$E\left[\sum_{i=1}^{n} N_i^2\right] = n * 1 + (n^2 - n) * 1/n \leq 2n \tag{18.2}$$

$\square$

This concludes the proof that general probabilistic algorithms can sort nicely distributed data in linear time.

Chapter 19

Finding the median

19.1 Introduction

Consider the problem of finding the median of n elements, *i. e.* the element with order number $n/2$. The most straightforward method is to sort the elements and then choose element number $n/2$. As sorting has complexity $O(n \log n)$ the time required for finding the median is also $O(n \log n)$.

However, the information we want is just the element with order number $n/2$ and we are not really interested in the order among the larger or smaller elements. Algorithms for median finding needing only $O(n)$ comparisons are available and we describe a number of such algorithms here. We start with a simple randomized algorithm.

19.2 The algorithm QuickSelect

Inspired by quicksort we propose the following randomized algorithm, called $\textsc{QuickSelect}(S, k)$:

1. Choose a random x uniformly in S.
2. Let $S_1 = \{y \in S : y < x\}$ and $S_2 = \{y \in S : y > x\}$.
3. Return $\begin{cases} \textsc{QuickSelect}(S_1, k) & \text{if } |S_1| \geq k, \\ \textsc{QuickSelect}(S_2, |S_2| + k - |S|) & \text{if } |S_2| \geq |S| + 1 - k, \\ x & \text{otherwise.} \end{cases}$

When we analyze the complexity of the algorithm we assume that all elements in $|S|$ are different. If any members are equal, we get the answer quicker, but the algorithm is more cumbersome to analyze.

Let $T(n)$ be the expected time it takes to find the k'th element in S, given that $|S| = n$. The x, which we select uniformly at random, is the ith element in the ordering of the elements of S with probability $1/n$ for all i. We can use this to obtain an expression for $T(N)$ by conditioning on i:

$$T(n) = n - 1 + \sum_{j=1}^{n} \mathrm{E}[\text{extra time} \mid i = j] \Pr[i = j] \tag{19.1}$$

We observe that the extra time needed is

$$\mathrm{E}[\text{extra time}] = \begin{cases} T(n-i) & \text{if } 1 \le i \le k-1, \\ 0 & \text{if } i = k, \\ T(i-1) & \text{if } k+1 \le i \le n, \end{cases} \tag{19.2}$$

which gives us the expression

$$T(n) = n - 1 + \frac{1}{n}\sum_{i=1}^{k-1} T(n-i) + \frac{1}{n}\sum_{i=k+1}^{n} T(i-1) \tag{19.3}$$

The most simple way to solve this recurrence is to guess the correct solution. Since we are confident that we have a very efficient algorithm we try to prove that $T(n) \le cn$ by induction. When we substitute this into Eq. (19.3) we get

$$T(n) \le n - 1 + \frac{c}{n}\left(\sum_{i=1}^{k-1}(n-i) + \sum_{i=k+1}^{n}(i-1)\right) \tag{19.4}$$

If we can show that the right hand side of this equation is at most cn, we have also shown that $T(n) \le cn$. To obtain an upper bound on the right hand side remember that the sum of an arithmetic series is the number of terms times the average of the first and last terms and hence

$$\begin{aligned}
&\sum_{i=1}^{k-1}(n-i) + \sum_{i=k+1}^{n}(i-1) = \\
&\frac{1}{2}((k-1)(n-1+(n-(k-1))) + (n-k)(k+n-1)) = \\
&\frac{1}{2}(n^2 + 2nk - 2k^2 + 2k - 3n) \le \frac{1}{2}(n^2 + 2nk - 2k^2) \le \frac{3n^2}{4},
\end{aligned}$$

where the last inequality follows since $2(nk - k^2)$ is maximized when $k = n/2$. If we plug this into Eq. (19.4) we obtain the relation

$$T(n) \le n - 1 + \frac{c}{n} \cdot \frac{3n^2}{4} \le n\left(1 + \frac{3c}{4}\right). \tag{19.5}$$

Now provided $c \ge 4$ this is bounded by cn and hence we have proved

Theorem 19.1. *Quickselect uses, in the expected sense, at most* $4n$ *comparisons.*

We get a couple of immediate questions. The first one is whether the analysis is tight. The reason one might expect that it is not is that we have derived a bound independent of k and we would think that it is easier to find the maximum than to find the median. The analysis is indeed not tight and if $k = cn$ the expected number of comparison [31] is $2(1 - c\ln c - (1-c)\ln(1-c))n + o(n)$ and in particular for the median we get $(2 + 2\ln 2)n$.

The second question that arises is whether there are more efficient randomized algorithms. This turns out to be the case and we discuss one in the next section.

The final question is whether we need randomness to have an algorithm using only $O(n)$ comparisons. This turns out not to be the case and we end this section by describing a couple of deterministic time linear algorithms. For practical implementations, however, we recommend Quickselect or variants thereof.

19.3 Outline of a better randomized algorithm

The reason that Quickselect runs in linear time is that the random element x is often of "middle size" by which we mean that it is not too far from the median. In those cases the size of problem in the recursive call is decreased by a constant factor. To decrease the running time even further we would like to spend some extra effort that makes x even more likely to be close to the median. In many practical implementations one chooses three random elements y, z and w and lets x be the median of these 3 elements. Already this gives an improvement in the constant but to get the best constant we should be even more careful how to choose x. We therefore take the following as our first step.

1. Choose $r = n^{3/4}$ random elements R and let x be the median of R. Partition S with respect to x, i.e. construct $S_1 = \{y|y < x\}$ and $S_2 = \{y|y > x\}$.

Let us analyze this heuristically. The expected number of elements of R that are larger than the median of S is $r/2$ and the standard deviation of this number is $O(\sqrt{r}) = O(n^{3/8})$. Thus we expect that if we inserted the median of S into the ordered list of R that it would be $O(n^{3/8})$ positions away from x. Since one position in R corresponds to $n^{1/4}$ positions in S we expect x to be $O(n^{5/8})$ positions away from the median of S (counted as positions in S). This means that the larger of S_1 and S_2 will be of size $n/2 + k$ where $k = O(n^{5/8})$. Assume that S_1 is the larger set. The median is now the k'th largest element in S_1 and to give an intuition how to find such an element let us discuss how to find the second largest element among m elements.

Suppose that m is a power of two and we find the maximum by a tennis tournament. In other words we first make $m/2$ disjoint comparisons and then compare the winners in $m/4$ pairs. We then keep pairing up the winners until we have the largest element at the end. The second largest element must have lost to the largest and since the largest was compared to only $\log m$ elements we can find it in another $\log m$ comparisons. Extending this method we can find the k'th largest element in $m + (k-1)\log m$ comparisons. Applying this method we can find the k largest element in S_1 in $n/2 + O(n^{5/8}\log n)$ comparisons. Since the first step needs $n + o(n)$ comparisons we get a total of $3n/2 + o(n)$ comparisons. Although this argument is only heuristic it can be made exact but we omit the details. Details and an argument that 1.5 is the best possible constant can be found in [19].

Theorem 19.2. *The median can be found by a randomized algorithm which does on the average $1.5n + o(n)$ comparisons.*

We turn next to deterministic algorithms.

19.4 A simple deterministic algorithm for median in $O(n)$

The main ingredient in the probabilistic median finding algorithms was to find an element x that was close to the median. This is also the key in our first deterministic algorithm. The difference here is that we have to be sure to find an element that is fairly close to the median. It will cost us more and the result will not be as close to the median.

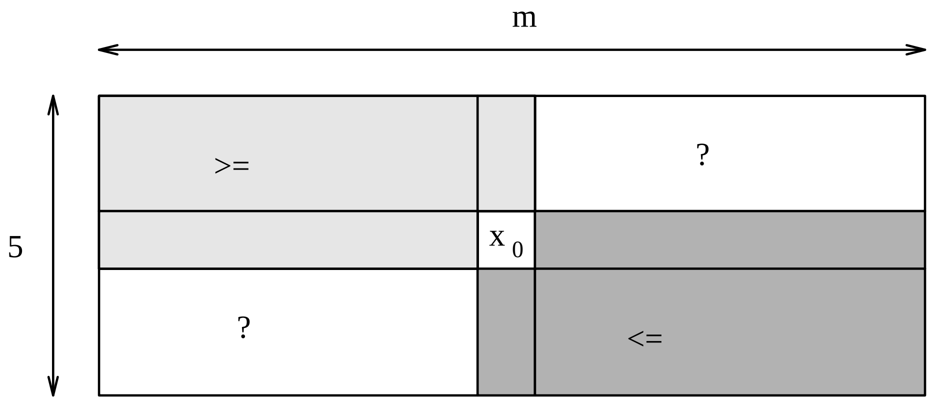

Figure 19.1: The two different shadings illustrate elements larger and smaller than x_0. We do not know how the non-shadowed elements are related to x_0.

19.4.1 Algorithm description

In this algorithm we assume that $n = 5m$.

Idea: Partition the n elements into m groups with 5 elements in each group. Find the median in each group. Find the median of the m medians from the 5-groups. Call this last median x_0.

The result so far is illustrated by the rectangle of figure 19.1 where the elements are arranged so that the 5-element groups are placed columnwise in the rectangle. The largest elements are at the top and to the left. The elements in the upper left of the rectangle must all be larger than or equal to x_0. Similarly, the elements in the bottom right are all smaller than x_0. What is achieved here is the first step of a partitioning of the whole set around the element x_0. We now partition the rest of the elements by comparing them with x_0. The set is now partitioned into two parts and the median must be in the largest of the two parts. We continue the search in this part but we no longer search for the median.

A more formal description for finding the median, or an element with order k, is the following.

Let $F(S, k)$ = find and return element of order k from set S. $|S| = n$. Assume $n = 5m$. If not, one group is a little bit smaller.

- Partition S into groups with 5 elements each.
- Let $S_1 = \{\text{medians from these groups}\}$, $|S_1| = m$
- $x_0 = F(S_1, m/2)$
- $S_2 = \{x | x \in S, x < x_0\}$ $S_3 = \{x | x \in S, x > x_0\}$ If $|S_2| \geq k$ then $F(S_2, k)$ If $|S_3| \geq n - k + 1$ then $F(S_3, k + |S_3| - n)$. Otherwise, the answer is x_0.

19.4.2 Algorithm analysis

$M(n)$ = the number of comparisons needed to find the kth element of n elements.

We need to find the median of 5 elements. This can be done with 7 comparisons.

$$M(n) \leq (n/5) \times (\text{number of comparisons for median of 5 elements}) + M(n/5) + (4n/10) + M(max(|S_3|, |S_2|))$$

The second term comes from finding the median of medians (element x_0 above). The third term is the number of comparisons to complete the partitioning, the number of uncertain elements in the figure. The last term is the continued search among the remaining elements. The number of elements left cannot be larger than $(7n/10)$ because both S_2 and S_3 have at least $(3n/10)$ elements (see figure 19.1) so the largest of these must have at most $(7n/10)$ elements.

$$M(n) \leq 7n/5 + M(n/5) + 2n/5 + M(7n/10)$$

$$M(n) \leq 9n/5 + M(n/5) + M(7n/10)$$

The solution of a relation of this type where

$$M(n) \leq cn + M(an) + M(bn)$$

and $a + b < 1$ is

$$M(n) \leq \frac{cn}{1 - (a + b)}$$

With $c = 9/5$, $a = 1/5$ and $b = 7/10$ we get

$$M(n) \leq 18n$$

This algorithm was created in 1972 by Blum, Floyd, Pratt, Rivest, and Tarjan [6]. Being a little bit more careful, the constant 18 can be improved significantly. However, we get a better constant by trying a completely different approach.

19.5 "Spider factory" median algorithm

19.5.1 Some preliminaries

We now describe an algorithm which has complexity $5n+o(n)$. It was developed in 1976 by Schönhage, Pippenger and Paterson [46].

This algorithm uses the term "spider factory" which is a device that from a set of unordered elements produces partial orders with $2k + 1$ elements where k elements are larger and k elements are smaller than one element m. In the graphic image of this partial order, m is pictured as the body of the spider and the other elements are connected to m with legs, each leg denoting that a relation between two elements has been established. Legs pointing upwards connect elements $> m$ and legs pointing downwards connect elements $< m$ to the body. (The desired result of any median finding algorithm is such a spider with $k = n/2$.)

The production cost of the spider factory is the following:

- Initial cost, before spider production can begin: $O(k^2)$

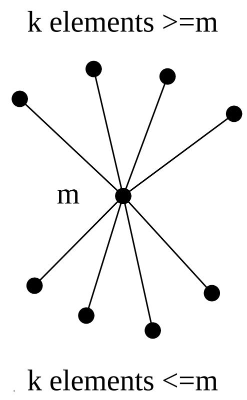

Figure 19.2: A spider with $2k+1$ elements

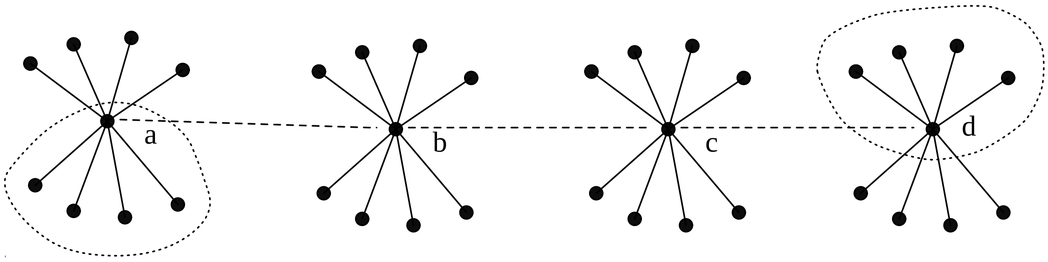

Figure 19.3: A list of 4 spiders sorted on the middle element, $a \leq b \leq c \leq d$.

- Residual cost, when no more spiders can be created: $O(k^2)$
- Cost per output spider: $5k$

A suitable choice of k is $k \sim n^{1/4}$. Details of the spider factory are given in section 19.5.4.

19.5.2 Algorithm

Run the spider factory until no more spiders are produced and arrange the spiders in a list sorted with respect to the middle elements of the spiders. Let n be the total number of elements and let r be the number of residual elements, *i. e.* the number elements that are left when all possible spiders are produced. The number of spiders in the spider list is then

$$s = \frac{n-r}{2k+1} = O(n/n^{1/4}) = O(n^{3/4})$$

Now, consider the lower legs and the body of the leftmost spider, S_l. These $k+1$ elements are all smaller than p_1 elements where $p_1 = k + (s-1)(k+1)$ (upper legs of leftmost spider and upper legs and body of the other spiders). Each of the elements in S_l could be larger than at most p_2 elements where $p_2 = sk - 1 + r$. As long as $p_1 > p_2$ the median cannot be one of the elements in S_l and these elements can be discarded. In the same way the elements of the upper legs and body of the rightmost spider must all be larger than the median and can be discarded. The remaining parts of these two spiders are returned to the factory and their elements used for building new spiders. Evaluating $p_1 > p_2$ gives the simple condition $s > r$. The procedure of

- chopping off partial spiders at ends of list
- returning remainders to the factory
- producing new spiders
- sorting spiders into list

successively reduces the number of remaining elements and the size of the list and is continued as long as $s > r$. When the procedure stops, run the simpler algorithm of section 19.4 on the remaining elements.

19.5.3 Analysis

Let t be the number of spiders produced. Then $t \leq n/(k+1) \sim n^{3/4}$ since for discarding $k+1$ elements at each end of the list, two spiders are used. Thus each spider allows us to eliminate $k+1$ elements and as at most n elements are eliminated the total number of spiders produced is at most $n/(k+1) \sim n/k = n/n^{1/4} = n^{3/4}$.

The cost of this algorithm has 4 terms

- Initial cost: $O(k^2) = O(n^{1/2})$
- Residual cost: $O(k^3) = O(n^{3/4})$
- Cost for producing the t spiders: $5tk = 5n^{3/4}n^{1/4} = 5n$
- Cost for inserting spider elements in the sorted list: $t \log s = O(n^{3/4} \log n^{3/4})$

The initial cost is the cost to start the spider production and is explained in the next section. The number of residual elements in the factory when spider production stops is $O(k^2)$, this also is shown in the next section. The total residual is then $O(k^3)$ because at this stage, $s \sim k^2$ and the remaining number of elements is $s(2k+1) + k^2 = O(k^3)$. The cost of finding the median of these remaining elements is $O(k^3)$.

The dominating cost here is producing the spiders and thus the overall cost for the algorithm for large values of n is $\sim 5n$.

Theorem 19.3. *The median of n elements can be found by $5n + o(n)$ comparisons.*

19.5.4 The spider factory

When creating the spiders a crucial idea is to compare the medium valued elements with each other and successively create partial orders which eventually can be used to construct the spiders that leave the factory. By "medium valued elements" we mean elements which have been found to be smaller than some number of elements and larger than some number of elements, these numbers differing with 0 or 1. The following procedure is used. For illustration, see figure 19.4.

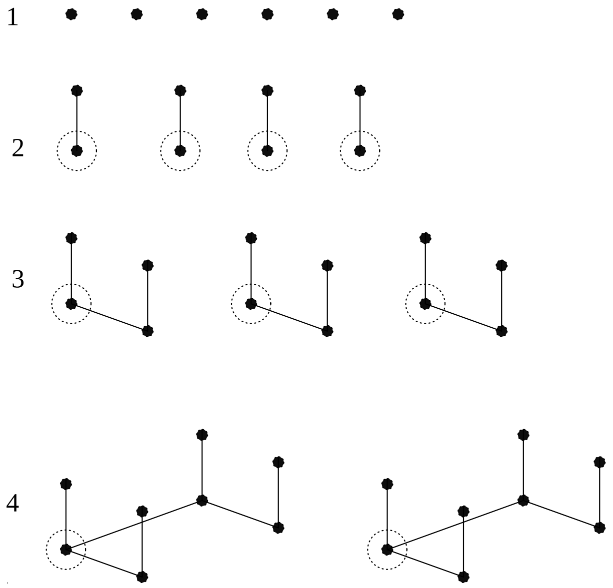

Figure 19.4: Illustration of steps $1, 2, 3$ and 4, creating partial orders of types H_0, H_1, H_2 and H_3.

- Begin with the unordered elements (instances of H_0).
- Compare elements pairwise and mark the smallest element, the "loser" in each pair (instances of H_1 are created).
- Compare the marked elements of the pairs and combine pairs into groups of 4 elements. This time we mark the largest element, the "winner" of each comparison (instances of H_2 are created).
- The marked elements of the quadruples are compared, the losers are marked and groups of eight elements are formed (instances of H_3).

These steps are continued so that

- Each group H_{2i} is built from two groups H_{2i-1} and the winners in the comparisons are marked.
- Each group H_{2i+1} is built from two groups H_{2i} and the losers are marked.

Lemma 19.4. *H_j contains 2^j elements.*

Proof. Obvious from the construction of H_j $\square$

Lemma 19.5. *An instance of H_{2l} with center element v contains at least $2^l - 1$ elements which are $> v$ at least and $2^l - 1$ elements which are $< v$ and thus a spider with $k = 2^l - 1$ can be output from H_{2l}.*

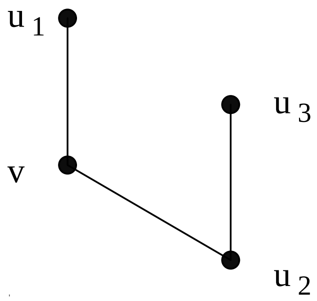

Figure 19.5: The center elements of 4 instances of H_{2p-2} used to build one instance of H_{2p} with center element v.

Proof. The proof is by induction. Suppose the lemma is true for $l = p - 1$ and show that this implies that it is true for $l = p$. An instance of H_{2p} is built from two instances of H_{2p-1} and these are built from four instances of H_{2p-2}. v is the center point of one of these. Let the other three center points be u_1, u_2 and u_3. See figure 19.5. The number of elements $> v$ are

- u_1 itself, 1 element
- The elements $> v$ in v:s instance of $H_{2(p-1)}$, $2^{p-1} - 1$ elements according to the assumption of the lemma.
- The elements $> u_1$ in its instance of $H_{2(p-1)}$, $2^{p-1} - 1$ elements also according to the lemma.

Summing the three terms we get the total number of elements $> v$

$$1 + 2^{p-1} - 1 + 2^{p-1} - 1 = 2^p - 1.$$

In the same way, we can count the number of elements of H_{2p} that are $< v$ by using $<$ instead of $>$ and u_2 instead of u_1. We get the same result, $2^p - 1$. We have shown that if the lemma is true for $l = p - 1$, then it is also true for $l = p$. It is very easy to check that the lemma is true for some small numbers, $l = 0, 1, 2$ and therefore it is true for all values of l. □

The spider factory builds instances of $H_0, H_1, \ldots, H_{2l-1}$ and whenever 2 instances of H_{2l-1} are available, an instance of H_{2l} is made and a spider with $k = 2^l - 1$ is output.

In order to estimate the complexity of the spider production we need to investigate what happens to the elements of H_{2l} that are not part of the spider and in particular we must look at how many of the edges within an H_{2l} are broken when the spider is created.

Creating the spider from H_{2l}, its center v is the center of the spider. We now look at the 4 instances of H_{2l-2} with the centers v, u_1, u_2, and u_3.

- $H_{2l-2}(u_3)$ just stays in the factory for further use, we cannot know how its elements are related to v.
- $H_{2l-2}(u_2)$: u_2 and all elements that we know are smaller go to the spider, the rest stays in the factory.

- $H_{2l-2}(u_1)$: u_1 and larger elements go to the spider, the rest stays in the factory.
- $H_{2l-2}(v)$: v is the body of the spider, elements known to be larger or smaller are in the spider.

The cost of the production is the number of comparisons, which is equivalent to the number of edges between elements. We count the edges when they leave the factory as part of a spider and when they are destroyed in the spider production. Each spider leaving the factory has $2k$ edges. The edges lost are $F_>(i)$ for removing from H_{2i} the center element and elements larger than the center element and $F_<(i)$ for edges lost when removing smaller elements. $F_>(i)$ and $F_<(i)$ can be expressed recursively as

$$F_>(i) = 1 + 2F_>(i-1), \qquad F_>(1) = 1$$

and

$$F_<(i) = 2 + 2F_<(i-1), \qquad F_<(1) = 2$$

and the solutions are

$$F_>(i) = 2^i - 1$$

and

$$F_<(i) = 2 \times 2^i - 2$$

The cost of removing both larger and smaller elements is

$$F_{>,<}(i) = F_{>,<}(i-1) + F_>(i-1) + F_<(i-1) + 1, \qquad F_{>,<}(1) = 1$$

$$F_{>,<}(i) = F_{>,<}(i-1) + 3 \times 2^{i-1} - 2 =$$

$$F_{>,<}(1) + \sum_{j=1}^{i-1}(3 \times 2^j - 2) =$$

$$1 + 3(2^i - 1) - 2(i-1) \leq 3(2^i - 1)$$

The cost for the lost edges is $3(2^i - 1) = 3k$ per spider. To get the total cost for each produced spider we add the cost for the output edges, $2k$, and we get $5k$.

We can now specify the terms of the total cost of the algorithm:

- Cost per produced spider: $5k$.
- Initial cost, the production of the first $H_{2l} \sim 2^{2l} \sim k^2$.
- The maximum residual in the factory should be one instance of each of $H_0, H_1, \ldots, H_{2l-1}$ because whenever 2 instances of H_j exist, they are combined into a H_{j+1}. As H_i has 2^i elements the residual have at most $\sum_{i=0}^{2l-1} 2^i \leq 2^{2l} \sim O(k^2)$ elements. The cost for calculating the median of the residual is then $O(k^2)$.

Another, more complex, algorithm along the same lines from 1976 by the same authors has complexity $3n + o(n)$ and a very recent (1994) modification of this algorithm by Dor and Zwick [15] has complexity $2.95n + o(n)$. A lower bound is $2n - o(n)$ and was obtained 1985 by Bent and John. Also this bound was improved by Dor and Zwick [16] to $(2+\epsilon)n - o(n)$ for very small (around 2^{-30}) ϵ.

Chapter 20

Searching

We now assume that we have a fixed set of elements $x_1, x_2, \ldots x_n$ and we repeatedly want to determine whether different elements x appear in this set. Thus time can be spent to preprocess the set to make later searches efficient. The standard form of preprocessing is sorting but later we will also consider other methods.

In a comparison-based computation model it is not difficult to show the number of comparisons needed to find some element x is $\log_2 n$. This follows since we have n different answers and thus need these many different outcomes of our set of comparisons. This bound is actually tight, since binary search finds an element in at most $\lceil \log_2 n \rceil$ comparisons and thus there is little to add in this model.

We next consider two other methods. The first is almost comparison based and is more efficient in the case when the inputs are random while to other is based on hashing.

20.1 Interpolation search

The underlying principle of interpolation search is that if we are trying to locate .023612 in a set of 1000000 elements which are random numbers from $[0, 1]$ it makes little intuitive sense to compare it to x_{500000} since that element is probably close to .5 and thus we are pretty sure that our element will be among the first half. It make a lot more sense to compare it to x_{23612} since this is the approximate place we expect to find .023612. The same idea is then applied recursively and we each time interpolate to find the best element to use for the next comparison. The algorithm maintains a left border ℓ and a right border r and works as follows.

1. Start with $\ell = 0$, $r = n + 1$, $\ell_v = 0$, and $r_v = 1$.

2. Set $g = \lceil \ell + (x - \ell_v)(r - \ell)/(r_v - \ell_v) \rceil$. If $g = r$ change g to $r - 1$.

3. If $x_g < x$, set $\ell = g$ and $\ell_v = x_g$. If $x_g > x$, set $r = g$ and $r_v = x_g$. If $x_g = x$ we are done. If $r - \ell \leq 1$ report failure.

4. Repeat from step 2.

To analyze this algorithm rigorously is rather delicate and we are content with a heuristic argument.

The correct measure to study is the expected number of positions between the sought element and the closes of the two borders r and ℓ. Assume this distance is m at some point. Look at the random variable which is that actual number of positions between the sought element and the closest border. By definition the expected value of this random variable is m and the standard deviation is $O(\sqrt{m})$. This means that the guess g will be around $O(\sqrt{m})$ positions wrong. This indicates that the measure of progress will go from m to $O(\sqrt{m})$ in one iteration and hence to $O(m^{1/4})$ in two iterations etcetera. Since the measure is at most n initially, it will be roughly $n^{2^{-i}} = 2^{2^{-i}\log n}$ after the i'th iteration. We conclude that after roughly $\log\log n$ iterations the correct position should be found. This is in fact correct and for the details we refer to [23]. Even though we have not done the complete proof we take the liberty of stating the theorem.

Theorem 20.1. *There is an absolute constant D such that interpolation sort, when applied to data uniformly distributed in $[0,1]$ has an expected number of iterations which is $D + \log\log n$.*

20.2 Hashing

The setting is that we have elements x_1, x_2, $\ldots x_n$ from some universe U. We want to store it in a table which has room for m elements. We have a *hash function* h, and for each x_i, $h(x_i)$ determines where in the table we are to store x_i. If we have many x_i mapping to the same place in the table there are a number of alternatives on how to proceed but we assume here that we let the entry in the table be the head of a linked list in which we store the elements. This implies that access time is proportional to the number of elements that map onto a given position and the key is to keep this number small.

The weakest demand we can have on h is that it should spread the elements uniformly in the table, *i. e.*

$$\forall i \Pr_x[h(x) = i] = 1/m, \tag{20.1}$$

where the probability is taken over a random input. As we have stated a number of times before it is not so satisfactory to rely on the randomness of the input and hence we should seek other solutions when possible and this is the case here.

Instead of having a fixed function h we will have a family, H, of possible functions and we want our estimates to be true, not for a random input, but for *every* input sequence and over the choice of a random $h \in H$. The first condition one would ask for is that each element is mapped to a random position.

$$\forall x, i \Pr_{h\in H}[h(x) = i] = 1/m. \tag{20.2}$$

This is good but clearly not sufficient since the family H consisting of the m constant functions satisfy this property and they are obviously useless as hash functions. The key condition is obtained from looking at pairs of elements and asking that they are mapped independently of each other.

Definition 20.2. *Let H be a family of hash functions. Then, H is a family of* pairwise independent *hash functions if*

$$\forall x_1 \neq x_2, i_1, i_2 \Pr_{h \in H}\big[h(x_1) = i_1 \cap h(x_2) = i_2\big] = \frac{1}{m^2}. \tag{20.3}$$

Functions satisfying the property above are also called *universal hash functions.* They were introduced by Carter and Wegman [9] and are of great importance, both for theoretical investigations and for practical implementations.

Example 20.3. *Let $U = \mathbb{Z}_2^\ell$ and $m = 2^k$. Consider functions of the form*

$$h(x) = Mx + b, \tag{20.4}$$

where M is a $k \times \ell$ matrix with elements in $\mathbb{Z}_2$ and $b \in \mathbb{Z}_2^k$. We let H be the set of functions obtained by varying M and b in all possible ways. Then in fact H is a set of universal hash functions. Namely consider $x_1 \neq x_2$ from U and take any $i_1, i_2 \in \mathbb{Z}_2^k$. We claim that the probability that $h(x_1) = i_1$ and $h(x_2) = i_2$ is exactly 2^{-2k}. Let us sketch a proof of this.

Since x_1 and x_2 are different there is some coordinate which is different and let us for concreteness assume that the first coordinate of x_1 is 0 and the first coordinate of x_2 is 1. First note that just by the existence of b in the definition of h, $Pr[h(x_1) = i_1] = 2^{-k}$ for any i_1. Now we want to estimate the probability that $Pr[h(x_2) = i_2]$ conditioned upon this. Since the first coordinate of x_1 is 0, $h(x_1)$ is not dependent on the first column of M. Furthermore since the first coordinate of x_2 is 1, when the first column of M varies over all 2^k possible values $h(x_2)$ varies over all 2^k possible values. It follows that the probability that $h(x_2)$ takes any particular value, even the dependent on $h(x_1)$ taking a particular value is 2^{-k} and thus we have a set of universal hash functions.

Example 20.4. *Let p be a prime which is greater than all $x \in U$. Choose a, b independently and uniformly at random from $\mathbb{Z}_p$ and set*

$$h(x) = (ax + b \bmod p) \bmod m. \tag{20.5}$$

We claim that this almost specifies a set of universal hash functions. Let us argue this informally. Fix x_1 and x_2. We claim that the values of $ax_1 + b$ and $ax_2 + b$ when we vary a and b over all possible choices this pair takes all p^2 values exactly once. To see this note that the system

$$ax_1 + b \equiv y_1 \bmod p \tag{20.6}$$

$$ax_2 + b \equiv y_2 \bmod p \tag{20.7}$$

has one unique solution (a, b) for any pair (y_1, y_2). This follows since the determinant of the system is $x_1 - x_2$ and since $x_1 \neq x_2$ and $p \geq \max(x_1, x_2)$ this is nonzero modulo p. Since, for each pair (y_1, y_2), there is exactly one pair of values for a and b, we have that $h(x_1)$ and $h(x_2)$ are independent over randomly chosen a and b. Thus,

$$\Pr\big[h(x_1) = i_1 \cap h(x_2) = i_2\big] = \Pr\big[h(x_1) = i_1\big] \Pr\big[h(x_2) = i_2\big]. \tag{20.8}$$

Now, write p as $cm + d$, where $d < m$. Given an x, we want to compute $\Pr_{a,b}\big[h(x) = i\big]$. If $i \leq d$, this probability is $(c+1)/p$, but if $i > d$, the probability *is c/p. Thus, $\Pr_{a,b}\bigl[h(x) = i\bigr]$ is not the same for all i, which implies that h does not fulfill the requirements of a universal hash function. The difference between $\Pr_{a,b}\bigl[h(x) = i \bigm| i \le d\bigr]$ and $\Pr_{a,b}\bigl[h(x) = i \bigm| i > d\bigr]$ is $1/p$ and if p is large this is very small, and we consider h as an "almost" universal hash function.*

We now collect a couple of simple facts on universal hash functions.

Theorem 20.5. *For a family H of universal hash functions, for any two $x_1, x_2 \in U$, $\Pr_{h\in H}\bigl[h(x_1) = h(x_2)\bigr] = 1/m$.*

Proof. Since

$$\Pr_{h\in H}\bigl[h(x_1) = h(x_2)\bigr] = \sum_{i=1}^{m} \Pr_{h\in H}\bigl[h(x_1) = i \cap h(x_2) = i\bigr] = \frac{1}{m}, \tag{20.9}$$

the probability that two elements collide is $1/m$, for any two elements in U. $\square$

Corollary 20.6. *If we use a random h from a family of universal hash functions, with $m \ge n$, the expected time to find an element x is $O(1)$. This is true for any set of n inputs.*

Proof. We just have to check the expected length of the linked list in which we store x is $O(1)$. By the previous corollary the probability that any other fixed elements is mapped to the same list is $1/m$. Since there are $n-1$ other elements the expected number of elements in the list is at $(n-1)/m \le 1$. $\square$

We emphasize that the above corollary is true for any fixed element in any list and the randomness is only over $h \in H$. If we fix a choice h and look among the n different elements we cannot, however, hope that all elements map to lists of constant length. For an average h it will be true for most elements however. Next we investigate the possibility of no collisions.

Corollary 20.7. *The expected number of collisions is $n(n-1)/2m$.*

Proof. There is a total number of $\binom{n}{2}$ pairs of n elements. The probability that each pair collides is $1/m$. The corollary follows by the linearity of expectation. $\square$

What does this last result imply? We study two candidates:

Corollary 20.8. *For $m > n^2/2$, there exists a "perfect" hash function,* i.e. *a hash function without any pair of elements colliding.*

Proof. This actually follows without too much thought. Namely, if $m > n^2/2$, the expected number of collisions is less than 1. This implies that it must exist some hash function which is collision free. $\square$

Conjecture 20.9. *For $m < n^2/2$ often have collisions.*

This does not follow from the results above, since the expected value gives us no information on how the collisions occur. It seems likely that the conjecture is true, but to really prove it, we would have to show something like $\Pr[\text{no collision}] \le 2m/n^2$ or $\Pr[\text{no collision}] \le e^{-n^2/(2m)}$. From all we know upto this point it could be that we have very many collisions a small fraction of the time (giving the large expected value) while most of the time there are no collisions. We do not enter into such details of hashing since the situation gets complicated. We simply state (without proof) that for most families of hash functions the conjecture seems, based on practical experience, to be true.

20.3 Double hashing

The problem we want to address in this section is the storage of n elements of data in a memory of size $O(n)$ and still having an access time that is $O(1)$ in the *worst* case. From standard hashing we could only get that the time was $O(1)$ for many, or even most, elements. The solution to this problem is to use the idea twice and it is therefor called *double hashing* . Fix a set H of universal hash functions.

First we use a random $h \in H$ with $m = n$. This creates a set of linked lists and assume that N_i is length of list i. Since

$$\sum_{i=1}^{n} \frac{N_i(N_i - 1)}{2} \tag{20.10}$$

is the numbers of pairs of elements that map to the same list, we know, by Corollary 20.7 that the expected value of this number is $n^2/2m$ which in our case is $n/2$. Thus at least half of all $h \in H$ have this number at most n. Fix one such h. Note that h can be found efficiently since we just keep picking random h and calculating the number of collisions. Once we get one with at most n collisions we keep it. The expected number of retries is constant. For all the linked list that are of size at most 3 (any other constant would do equally well) we just keep the linked list, while for other values of N_i we are going to replace the linked list by a second level of hashing. This time without collisions. Given any list L_i of length N_i we choose and m_i with $m_i > N_i(N_i - 1)/2$ and find a hash function h_i mapping U to $[m_i]$, which exists by Corollary 20.8, without collisions[1]. We store the description of h_i for the head of the linked list and then use m_i words of storage to store the items in L_i. Most of these words will be empty but this is of no major concern (at least in theory).' We only need the fact that

$$\sum_{i=1}^{n} m_i \leq \sum_{N_i > 3} 1 + \frac{N_i(N_i - 1)}{2} \leq \frac{7n}{6}$$

where we used the fact that (20.10) is at most n and since each term that we consider is at lest 6, there are at most $n/6$ terms and thus the extra 1's add up to at most $n/6$.

Now we claim that the worst case access time in this data structure is $O(1)$. To see this fix any x and consider $i = h(x)$. Either the entry i is the header for a linked list of size at most 3 or it is a description of a hash function h_i. In the latter case we compute $h_i(x)$ and we find x in the small hash table.

[1]We might want to double m_i if we have trouble finding the h_i quickly. This depends on the parameters in question and the constants involved. Since we only care about $O(n)$ storage, doubling m_i only affects the constant but not the result.

Chapter 21

Data compression

The basic problem is to compress a long string, in order to get a shorter one. For a compression algorithm A we denote the result of A applied to x by $A(x)$. We want to be able to recover the original information and hence we have a companion decompression algorithm with a corresponding function $D_A(x)$. We have two types of compression algorithms.

- **Lossless** algorithms that satisfy $D_A(A(x)) = x$ for all x.
- **Lossy** algorithms where $D_A(A(x))$ is only close to the original x. Close is defined in a suitable way depending on what x represents.

Lossy algorithms are usually applied to items such as pictures while for text and string information we usually want lossless algorithms. We assume throughout this chapter that compression algorithms are self-delimiting in that there is no end-of-file marker that is used. Each algorithm knows by itself when to stop reading the input. This implies in particular that there are no inputs x_1 and x_2 such that $A(x_1)$ is a prefix of $A(x_2)$.

21.1 Fundamental theory

Given a fixed string x, how can it best be compressed? In this situation there can exist special purpose algorithms and that do extremely well on this particular input and thus one might fear that studying the compression of a single string is meaningless. In particular for each string x the there is a lossless compression algorithm A_x which compresses x to a single bit. The key to avoiding this problem is to measure also the size of the compression (or to be more precise decompression) algorithm. Since any input to the decompression algorithm can be incorporated into the source code we can simply study programs without input and look for programs that output x without input.

Definition 21.1. *The* Kolmogorov complexity, *$K(x)$ is the size (in bits) of the smallest inputless program that outputs x.*

This is not a mathematically strict definition in that we have not specified the meaning of “program”. It seems like if we consider programs written in Java we get one function K_{Java} while using programs written in C we get a different function K_C. Looking at more formal models of computation like a Turing machine we get a third function K_{TM} which might behave rather differently compared to the first two. Let us first compare the first two notions.

Theorem 21.2. *There are two absolute constants A and B such that for any string x we have $K_{Java}(x) \leq K_C(x) + A$ and $K_C(x) \leq K_{Java}(x) + B$.*

Proof. We prove only the second inequality, the first being similar. We claim that we can write a "Java simulator" in C. This program takes as input a text string and checks if it is a correct Java program that does not take any input. If the answer to this is affirmative the program simulates the actions of the Java program and gives the same output as the Java program would have produced. We hope the reader is convinced that such a program exists. Suppose the size of this program is S. Now take any program of size K written in Java. By hardwiring the program into the above simulator we can create a C-program of size $K + S'$[1] with the same output. By choosing the shortest program in Java that outputs x we can conclude that $K_C(x) \leq K_{Java}(x) + S'$ and the theorem follows by setting $B = S'$. □

Note that the above proof does not use anything specific of C and Java but only the property that they are universal programming languages that can simulate each other. Thus the theorem can be generalized to prove that many notions of "program" give rather closely connected variants of Kolmogorov complexity. This remains true even when using formal models of computation such as a Turing machine but the details of such simulations get cumbersome and we omit them.

The moral of Theorem 21.2 together with its generalizations is that the choice of notion of "program" has little importance for $K(x)$ and the reader can choose his favorite programming language to be used in the definition of Kolmogorov complexity. For the duration of these notes we leave this choice undefined in that we, in any case, only specify programs on very high level.

Let us state the immediate application to data compression.

Theorem 21.3. *For any lossless compression algorithm A :*

$$Length(A(x)) \geq K(x) - C_A,$$

where C_A is a constant that depends only on A.

Proof. A program that outputs x is given by the decompression algorithm D_A together with $A(x)$. The size of this program is essentially the size of $A(x)$ plus the size of the code for D_A and thus the total size is $Length(A(x)) + C_A$ for some constant C_A that depends only on A. Thus we have put our hands on one small program that outputs x and by the definition of Kolmogorov complexity we have $K(x) \leq Length(A(x)) + C_A$ and the theorem follows. □

Theorem 21.3 gives a bound on how much we can compress a string x. It turns out that the bound is mainly of theoretical interest. The problem being that $K(x)$ is not computable.

[1]One could hope that the size of this program would be bounded by $K + S$ but this might not be the case due to silly details. In particular the statement of reading an input (like *gets*(s);) need to be replaced by an assignment *strcpy*(s,"..."); and we might get a different additive term than the size of the string we are using in the assignment.

Theorem 21.4. *K(x) is not a computable function.*

Proof. Before we attempt a formal proof let us recall a standard "paradox". Consider describing integers by sentences in the English language. We allow descriptions like "ten" but also "the smallest integer that can be written as the sum of two cubes of positive number in two different ways"[2]. Clearly some integers require very long description since there are only 26^l different descriptions of length l. Now consider the description "the smallest integer that cannot be described with less than one thousand letters". There must be such an integer, but the problem is that now we have a description with less than 1000 letters for this very integer and thus we appear to have a paradox. The problem when trying to present this paradox in a formal setting is how to capture the informal notion of "description". One mathematically precise way would be to let a description of x be a program that outputs x and thus the minimal length description of x has length $K(x)$. The above integer would then be the smallest y such that $K(y) \geq 1000$. The problem this is not a description in the form of a program outputting y. *If* Kolmogorov complexity was a computable then it would be easy to turn this description into the required program outputting y and we would have a true paradox. As we are about to prove this is not true and hence there is no paradox. Let us now turn the formal proof of the theorem.

Assume for contradiction that K can be computed by a program of size c. Consider the following program.

```
for i=1,... do
       if (K(i)) > 2c +1000 then output i and stop
    od
```

Using the assumed subroutine for K we can write a program of size slightly larger than c (to be definite of size at most $c + \log c + 500$) executing the high level routine above. The $\log c$ bits are needed to specify the number c. Consider the output of this program. Assuming that we have a subroutine that correctly computes K it must be an integer i such that $K(i) \geq 2c + 1000$. On the other hand since the program is of size $c + \log c + 500$ we have, by definition of Kolmogorov complexity, that $K(i) \leq c + \log c + 500$. We have reached a contradiction and thus we conclude that the cannot exist a program that computes Kolmogorov complexity. □

The above theorem just says that $K(x)$ cannot be computed exactly and thus it does not exclude the possibility of a program P such that for any x it is true that $K(x)/2 \leq P(x) \leq K(x)$. An easy variant of the above proof shows that also this P cannot exist and in fact it is possible to prove much stronger theorems on the inapproximability of K. We leave the details to the interested reader.

As a final conclusion of this chapter let us just note that although Kolmogorov complexity is a nice theoretical basis for data compression it is of little practical value since it, or any function close to it, cannot be computed. In our next section we turn to the question of compressing strings which are formed by some random process.

[2] A description of 1729

21.2 More applicable theory

In many situations it is useful to view the input as coming from a probability distribution. Two such examples are

- random strings of given length
- texts in English of given length.

In the first example we have a probability distribution that is mathematically easy to describe while in the second situation we do not. In either case, however, we have a random variable X and for each possible value x of X we have an associated probability p_x of seeing this value. In the case of random strings p_x is given by a closed formula and in the case of English texts the value exists although we have problems of finding the exact value.

In a similar way we can view pictures of various origins as probability distribution and we are interested in a compression algorithm that compresses a random value of X efficiently. The natural number that measures the success of a compression algorithm in this situation is to look at the expected size of the compressed string.

Definition 21.5. *For a compression algorithm A and probability distribution X let $E_A(X)$ the expected length of the output of A when fed a random value of X as input. In other words, if the probability that X takes the value x is p_x then*

$$E_A(X) := \sum_x p_x \cdot Length[A(x)].$$

It turns out that possible behavior of $E_A(X)$ can, in many situation, be estimated quite well. We have the following key definition.

Definition 21.6. *The* entropy *of a random variable X is denoted by $H(X)$ and defined by*

$$H(X) := -\sum_x p_x \log p_x.$$

Observe that entropy is a property only of the random variable X and that it is a real number. As an example consider a random variable that takes t different values each with probability $1/t$. A straightforward calculation shows that X has entropy $\log t$. On the other hand we the t different outcomes can be coded as $1, 2 \dots t$ and thus X can be coded with $\log t$ bits. It is not difficult to see that we cannot do better in this case and also in general there is a relation between entropy and code length.

Theorem 21.7. *For any compression algorithm A and any random variable X we have*

$$E_A(X) \geq H(X).$$

Proof. Suppose x is coded with l_x bits. Then by basic properties of prefix-free codes we have

$$\sum_x 2^{-l_x} \leq 1. \tag{21.1}$$

We omit the proof of this fact but encourage the interested reader to prove it. We have that

$$E_A(X) = \sum_x p_x l_x \tag{21.2}$$

and thus we want to find the minimal possible value of this sum under the constraint (21.1). Note that we are here treating p_x as given constants while we try to find the optimal values of l_x for all x. By standard analysis we know that for the optimum of such constrained optimization the gradient of the objective function is parallel to the gradient of the constraint. Since the derivative of 2^{-l_x} is $-\ln 2 \cdot 2^{-l_x}$, while the gradient of the objective function is given by the probabilities p_x, the optimal point satisfies $\ln 2 \cdot 2^{-l_x} = \lambda p_x$ for some parameter λ. Since $\sum_x p_x = 1$ and $\sum_x 2^{-l_x} = 1$ for the optimal point we have $\lambda = \ln 2$. This gives $l_x = -\log p_x$ and

$$\sum_x p_x l_x = -\sum p_x \log p_x = H(X)$$

is the desired optimum. Hence this is a bound on the performance on any compression algorithm and the theorem follows. □

The above theorem gives a good lower bound for $E_A(X)$ and let us next turn to upper bounds. If we know the value of p_x then the above proof suggests that we should code x with $-\log p_x$ bits. If this number is an integer for all x then it is always possibly to create such a code and we have a tight result. In general, the optimal prefix-free code can be constructed as the Huffman-code and it can be proved that we lose at most an additive 1.

Theorem 21.8. *The Huffman-code of a probability space X has an expected code length that is at most $H(X) + 1$.*

We omit the proof of this theorem.

Note that this is *not* a symbol-by-symbol Huffman-code but it is a gigantic Huffman-code treating each possible string as a different symbol. Thus this is not very useful for the obvious reason that any construction procedure would have to enumerate all different possible inputs and this is, in most cases, not feasible. In particular, if x is an English text of length 10000 characters we have 26^{10000} possible x to consider. On top of the obvious problem of too many possible x we have the additional problem of not knowing the explicit values of p_x.

One approach in practice is to break the input into pieces of fixed size (e.g. 3) and then to code each piece separately. In this case the relevant probability space is the set of all trigrams (segments of 3 letters). The probabilities of all trigrams can be estimated by doing frequency counts on long texts. Note, however, that this makes the performance sensitive to the type of text. If we have constructed a Huffman-code based on English text taken from a typical book this algorithm would probably perform poorly if we try to compress source code or a text in Swedish. Our next attempt is therefor to discuss algorithms which do not need any information on the input space but still performs well in theory and practice.

21.3 Lempel-Ziv compression algorithms

Ziv and Lempel invented two compression algorithms of rather related nature around 20 years ago. Many variations of these algorithms have been proposed since then, but since we are only interested in the basic ideas we describe here only the simplest variants of the algorithms. The algorithms have the following basic properties.

- They do not need any information on the input space.
- They are provably "optimal" in many simple situations. In particular, if the strings being compressed are obtained from independent random symbols, the compression ratio is asymptotically optimal.
- They compress most occurring strings significantly.
- They are fast and easy to implement.

Before we proceed, let us just remind the reader that there are some strings that are not compressed at all. This is true for any lossless compression algorithm since lossless compression algorithms are permutations of strings and thus if some strings are mapped to shorter strings some other strings must be mapped to longer strings. We now proceed to discuss the two algorithms which, due to their years of publication are known as LZ77 and LZ78, respectively.

21.3.1 LZ77- The sliding window technique

We introduce the technique by an example. Suppose we are in the middle of compressing a text and we are about to process the second "e" of the text shown below

	.	.	e	c	o	s	y	.	.		e	c	o	n	o	m	i	c

The algorithm locates the longest prefix that has appeared so far in the text. In this case this is "eco" which started, say, 27 positions earlier. The token (27,3,"n") is constructed which is the instruction to copy the string of length 3 that started 27 positions to the left and then add the symbol "n". This way "econ" is coded and the algorithms proceeds to find the longest matching of "omic.." etc.

In theory we allow matchings of arbitrary length from anywhere earlier in the text. In practice we limit how far back in the text we search for matchings (i.e. limit the buffer size) and also limit the length of the matching. As an example if we limit the buffer size to 2048 and the size of the matching to be at most 8 character we can code the two integers using 14 bits. There are many variants of this algorithm and we refer to [44] for a discussion of some.

Let us point out that the algorithm allows the matching string to go right of the current position and thus in particular a string of many equal symbols is coded very efficiently. In particular, given a string of N a's.

a	a	$\cdots$	a

the first token is (0,0,"a") and the next is (1,N-2,"a") which codes the rest of a's.

Finally we remark that the algorithm is asymmetric in that decoding is more efficient than coding. The decoder needs only to keep the buffer of decoded text and copy the appropriate parts. The coder needs to find the required matchings.

21.3.2 LZ78 - The dictionary technique

The principle is the same as in LZ77 in that we are looking for matchings with previous text. In this case the matchings are located by storing substrings of the text in a dictionary.

The dictionary is either initialized as empty or containing all one-symbol strings. When encoding a subset of text we look for the longest prefix of the current text that is stored in the dictionary and then we add one symbol. The concatenated string (the part from the dictionary together with the extra symbol) is then stored in the next position of the dictionary. Thus if we are in the same position as in the previous algorithm we might have stored "eco" at position 291 in the dictionary and in this case "econ" would be coded as (291,"n"). The string "econ" is then stored in the first free slot of the dictionary.

This algorithms is a little bit more symmetric than LZ77 in that both the coder and the decoder have to maintain the dictionary. Only the coder, however, needs to find matching prefixes. The decoder again only copies appropriate strings.

In theory LZ78 uses a dictionary of unlimited size. In practice one usually has a fixed upper bound. When the dictionary is full one either works with this static dictionary or simply starts over again with the empty dictionary. Again there are a large number of variants and we refer to [44] for some of them.

21.4 Basis of JPEG

Just as a teaser let us describe JPEG in a few words. It is a lossy compression algorithm used to compress pictures. The pixels are divided in blocks of size 8×8 and the cosine transform of the 8×8 matrices given by pixel-values in blocks are calculated. The cosine transform is a real valued variant of the Fourier transform and has rather similar properties.

The result of a cosine transform is 64 real numbers but for efficient storage these numbers are rounded to integer multiples of predefined quantities. The resulting integers are then coded with Huffman like codes. We again refer to [44] for details.

Bibliography

[1] Adleman & Huang: *Primality Testing and two-dimensional Abelian Varieties over Finite Fields.* LNCS, vol 1512, Springer-Verlag, 1992.

[2] Adleman, Pomerance & Rumely: *On Distinguishing Prime Numbers from Composite Numbers.* Annals of Math. 117, 1983.

[3] D. Applegate, R. Bixby, V. Chvatal, and W. Cook *On the solution of traveling salesman problems.* Documenta Mathematica Journal der Deutchen Mathematiker-Vereinigun, ICM III, 1998, pp 645-656.

[4] S. Arora, *Nearly linear time approximation schemes for Euclidean TSP and other geometric problems.* Proceeding of 38th IEEE Symposium on Foundations of Computer Science, pp 554-563, 1997.

[5] L. Auslander and S. V. Parter. *On imbedding graphs in the plane.* J. Math. and Mech., 10:517–523, 1961.

[6] M. Blum, R. Floyd, V. Pratt, R. Rivest and R. Tarjan. *Time bounds for selection.* Journal of Computer and System Sciences, Vol 7, 1973, pp 448-461.

[7] K.-H. Borgwardt. *The Simplex method a probabilistic analysis*, Springer-Verlag, Berlin, 1987.

[8] E. Brickell *Breaking iterated knapsacks* Advances in Cryptology, Proceednings of Crypto 84, Springer Verlag 1985, pp 342–358.

[9] J. Lawrence Carter and Mark N. Wegman. *Universal classes of hash functions.* Journal of Computer and System Sciences, 18:143–154, 1979.

[10] N. Christofides *Worst-case analysis of a new heuristic for the traveling salesman problem* Report n 388, GSIA Carnegie-Mellon University, Pittsburgh, PA, 1976.

[11] Cohen & Lenstra: *Primality Testing and Jacobi Sums.* Mathematics of Computation 42, 1984.

[12] D. Coppersmith and S. Winograd. *Matrix multiplication via arithmetic progressions.* 19th Annual ACM Symposium on Theory of Computing, 1–6, 1987.

[13] T. Cormen, C. Leiserson, and R. Rivest: *Introduction to Algorithms.* MIT Press, 1990.

[14] W. Diffie and M. Hellman *Multiuser cryptographic techniques.* AFIPS Conference Proceedings, 45, 1976, pp 109–112.

[15] D. Dor and U. Zwick *Selecting the median.* Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms, 1995, pp 28–37.

[16] D. Dor and U. Zwick *Median selection requires $(2 + \epsilon)n$ comparisons.* Proceedings of the 37th IEEE Symposium on Foundations of Computer Science, 1996, pp 125–134.

[17] J. Edmonds *Paths, tress and flowers.* Canadian Journal of Mathematics, Vol 17, 1965, pp 449–467.

[18] L. Engebretsen, *An explicit lower bound for TSP with distances one and two,* Technical report 45, 1998, Electronic Colloquim on Computational Complexity. http://www.eccc.uni-trier.de/eccc/

[19] Robert W. Floyd and Ronald L. Rivest. *Expected time bounds for selection.* Communication of the ACM, 18(3):165–172, 1975.

[20] Gerver: *Factoring Integers with a Quadratic Sieve.* Math. Comp. (41) 1983.

[21] A. J. Goldstein. *An efficient and constructive algorithm for testing whether a graph can be embedded in the plane.* Graph and Combinatorics Conf., Contract No. NONR 1858-(21), Office of Naval Research Logistics Proj., Dep. of Math. Princeton U., May 16–18, 2 pp, 1963.

[22] Goldwasser & Killian: *Almost all Primes can be Quickly Certified.* STOC 1986.

[23] Gaston H. Gonnet, Lawrence D. Rogers, and J. Alan George. *An algorithmic and complexity analysis of interpolation search.* Acta Informatica, 13:39–52, 1980.

[24] J. Hopcroft and R. Tarjan. *Efficient planarity testing.* Journal of the ACM, 21:549–568, 1974.

[25] J. Håstad. *Tensor Rank is NP-Complete.* Journal of Algorithms, Vol 11, 644–654, 1990.

[26] Ireland & Rosen: *A Classical Introduction to Modern Number Theory.* Springer-Verlag 1990.

[27] D. Johnson and L. McGeoch *The Traveling Salesman Problem: A Case Study in Local Optimization.* in Local Search in Combinatorial Optimization, E. H. L. Aarts and J. K. Lenstra (eds), John Wiley and Sons, Ltd., 1997, pp. 215-310. Available electronically at http://www.research.att.com/d̃sj/papers.html.

[28] D. Karger and M. Levine *Finding maximum flows in undirected graphs seems easier than bipartite matching.* Proceedings of 30th Annual ACM Symposium on Theory of Computing, 1998, pp 69-78.

[29] A. Karatsuba and Y. Ofman *Multiplication of multidigit numbers in automata.* Doklady Akad. Nauk SSSR 145, 1962, pp 293-296 (in Russian).

[30] N. Karmarkar *A new polynomial-time algorithm for linear programming*, Combinatorica, 4, 1984, 373-395.

[31] D. Knuth *Mathematical analysis of algorithms* Proceeding of the IFIP congress 1971, Volume 1, North Holland, pp 19-27.

[32] Lenstra: *Factoring Integers with Elliptic Curves.* Annals of Math. (2) 1987.

[33] Lenstra & Lenstra: *The Development of the Number Field Sieve.* Springer Verlag, 1993.

[34] A. Lenstra, H. Lenstra, L. Lovász: *Factoring Polynomials with Rational Coefficients* Math. Ann. 261, pp 515–534, 1982

[35] S. Micali and V. Vazirani *An* $O(\sqrt{|V|}|E|)$ *algorithm for finding maximum matching in general graphs.* Proceedings of 21'st IEEE Symposium on Foundations of Computer Science, 1980, pp 17-27.

[36] G. Miller: *Riemann's Hypothesis and Tests for Primality.* 7th STOC.

[37] Montgomery: *Speeding the Pollard and Elliptic Curve Methods of Factorization.* Math. Comp. (48) 1987.

[38] Morrison & Brillhart: *A Method of Factoring and the Factorization of* F_7*.* Math. Comp. (29) 1975.

[39] Pollard: *A Monte Carlo Method for Factorization.* BIT 15, 1975.

[40] C. Pomerance: *Analysis and Comparison of Some Integer Factoring Algorithms.* Comp. Methods in Number Theory, Math. Centrum, Amsterdam, 1982.

[41] C. Pomerance: *A tale of two sieves.* Notices of the American Mathematical Society, Volume 43, Number 12, 1996, pp 1473-1485.

[42] M. Rabin: *Probabilistic Algorithms for Testing Primality.* J. Number Theory 12, 1980.

[43] Alastair I. M. Rae *Quantum Mechanics* 3rd ed, Institute of Physics Publishing, 1996.

[44] David Salomon. *Data compression, the complete reference*. Springer Verlag, 1998.

[45] J.J. Sakurai *Modern Quantum Mechanics* Addison-Wesley Publishing Company, Inc, 1994

[46] A. Schönhage, M. Paterson and N. Pippenger. *Finding the median Journal of Computer and System Sciences*, Vol 13, 1976, pp 184-199.

[47] A. Schönhage, V. Strassen. *Schnelle multiplikation grosser Zahlen* Computing Vol 7, 1971, pp 281-292.

[48] A. Shamir *A polynomial time algorithm for breaking the basic Merkl-Hellman Knapsack.* Advances in Cryptology, Proceedings of Crypto 92, Plenum Press 1983, pp 279–288.

[49] Peter W. Shor *Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer.* SIAM Journal on Computing, October 1997, Volume 26, Number 5, pp 1484-1509.

[50] Peter W. Shor *Fault-tolerant quantum computation.* Proceeding of 37'th Annual IEEE Symposium on Foundations of Computer Science, 1996, pp 56-65.

[51] Solovay & Strassen: *A Fast Monte Carlo Test for Primality.* SIAM J. on Comp. 6, 1977.

[52] V. Strassen. *Gaussian elimination is not optimal.* Numerische Matematik, 13, 1969.

[53] A C-C Yao *Quantum circuit complexity.* FOCS, 1993, pp 352–361.

Index