Department of Computer Science

L.J. Holleboom

Some aspects of computing, data security, and upcoming technologies

Master's Thesis

2002:01

Some aspects of computing, data security, and upcoming technologies

L.J. Holleboom

© 2003 The author and Karlstad University

This thesis is submitted in partial fulfillment of the requirements for the Masters degree in Computer Science. All material in this thesis which is not my own work has been identified and no material is included for which a degree has previously been conferred.

L.J. Holleboom

Approved, December 6th 2002

Opponent: Christer Andersson

Advisor: Simone Fischer-Hübner

Examiner: Donald Ross

Abstract

A number of concepts from theoretical computer science are discussed. These concepts allow to decide which problems can be solved and which problems cannot be solved. The ongoing process of miniaturization, which leads to ever faster computers, will necessarily come to an end when components approach the size of atoms. Two different technologies that possibly can play a role in computing in the future are quantum computers and DNA computers. The principles of these technologies and some of the possible impacts, as for example in cryptography are discussed.

Contents

1 Introduction **1**

2 Theory of computation **5**

2.1 Introduction 5

2.2 Finite automata 5

2.3 Regular expressions 10

2.4 The pumping lemma 13

2.5 Pushdown automata 14

2.6 Context free languages 17

2.7 Equivalence of PDAs and CFGs 19

2.8 The pumping lemma for context free languages 21

2.9 Turing machines 24

2.10 The Church-Turing thesis 27

2.11 The halting problem 29

2.12 Hilbert's tenth problem 32

2.13 Self-reference 35

2.14 Complexity classes 37

2.15 Summary and outlook 40

3 Quantum computing **43**

3.1 Moore's law . . . 43
3.2 More physical limitations . . . 48
3.3 Energy dissipation and reversibility . . . 49
3.3.1 Logical gates . . . 49
3.3.2 Universal gates . . . 49
3.4 How to compute quantum mechanically . . . 52
3.4.1 The classical bit . . . 52
3.4.2 The quantum bit . . . 52
3.4.3 Many qubits . . . 54
3.5 Quantum computation . . . 55
3.5.1 Single qubit gates . . . 55
3.5.2 Multiple qubit gates . . . 57
3.5.3 Function evaluation and quantum parallelism . . . 59
3.5.4 Deutsch's algorithm . . . 62
3.6 The Fourier transform . . . 64
3.6.1 Mathematical Definition . . . 64
3.6.2 The Discrete Fourier transform . . . 64
3.6.3 The Fast Fourier transform . . . 65
3.6.4 The Quantum Fourier transform . . . 67
3.6.5 Application of the quantum Fourier transform . . . 77
3.6.6 Factoring on a quantum computer . . . 82
3.7 A fast quantum search algorithm . . . 85
3.7.1 Closed form expressions for α_k and β_k . . . 91
3.7.2 Number of iterations . . . 93
3.8 Summary . . . 94
4 Cryptography and quantum key distribution **95**
4.1 Introduction and terminology . . . 95

4.2 The Vernam cipher 96
4.2.1 The security of the Vernam cipher 98
4.3 The key distribution problem 99
4.3.1 Preliminaries 99
4.3.2 A solution based on mathematical principles 100
4.3.3 The RSA cryptosystem 101
4.3.4 A solution based on physical principles 103
4.4 Quantum key distribution 107

5 Yet another direction: DNA computing 111

6 Conclusion and summary 115

References 117

A 121
A.1 Some details on Deutsch algorithm for parallel function evaluation 121
A.2 The discrete Fourier transform of a periodic function 122

List of Figures

2.1 *A simple finite automaton, that checks if the sum of the input digits is a multiple of* 3. 6

2.2 *The NFA corresponding to the regular expression* $\{ab, a\}^*$. 12

2.3 *Pushdown automaton with input and stack.* 15

2.4 *State diagram of the PDA accepting the language* $A = \{0^n 1^n | n \geq 0\}$. 17

2.5 *Parse tree for the string* 000111. 19

2.6 *State diagram of a PDA that accepts all strings generated by the CFG given in 2.2* 21

2.7 *Effect of pumping lemma on the parse tree.* 23

2.8 *Turing machine.* 25

2.9 *State diagram of a Turing machine recognizing the language* $B = \{1^{2^n} | n \geq 0\}$ 26

2.10 *Testing program correctness.* 28

2.11 *Hierarchy of languages.* 32

2.12 *Turing machine that writes out its own description* 35

2.13 *Turing machine that can obtain its own description* 36

2.14 *Probable relation between complexity classes.* 39

2.15 *Probable structure of the P and NP complexity classes.* 40

3.1 *Moore's law* 45

3.2 *Processor clock frequency* 47

3.3 *Reversible and irreversible gates* 50
3.4 *Toffoli gate: reversible and universal.* 51
3.5 *Toffoli gate is its own inverse.* 51
3.6 *qubit* 54
3.7 *NOT and Hadamard quantum gates* 57
3.8 *The CNOT quantum logic gate.* 58
3.9 *Function evaluation of a function on 1 bit.* 59
3.10 *Quantum parallelism.* 60
3.11 *Circuit determining balanced or constant.* 62
3.12 *Controlled* R_p *gate* 74
3.13 *Circuit performing the quantum Fourier transform* 75
3.14 *Swapping two qubits.* 75
3.15 *Logic circuit swapping two qubits.* 76
3.16 *Realization of* I_{x_0} *by means of* U_f. 88

4.1 *Wave packet* 104
4.2 *Electromagnetic wave* 106
4.3 *Encoding of bits* 108
4.4 *Decoding of bits* 108

5.1 *Graph of seven vertices* 113

List of Tables

2.1 *The transition function $\delta(r, s)$ for the finite automaton in figure 2.1.* 7

2.2 *Specification of the transition function for a PDA accepting $A = \{0^n 1^n | n \geq 0\}$.* 16

2.3 *Some complexity classes.* 38

3.1 *Transistor count per chip* 44

3.2 *Factor in quantum Fourier transform* 73

4.1 *The Vernam cipher* 97

4.2 *The Vernam cipher, text only* 97

Chapter 1

Introduction

The past four decades computers have become smaller, faster, and cheaper, a development which is expected to continue for at least one, possibly a few more decades. The underlying reason for this development is the miniaturization of electronic components. Computer industry has managed in sustaining an exponential increase of the number of transistors per chip, measured as a function of time. The increase has actually been quantified under what has come to be known as Moore's law, that the number of transistors per chip doubles approximately every eighteen months, while the size of the entire chip remains constant. With exponential decrease in size follows an exponential increase in computational power in the form of exponentially increased clock frequency. No doubt, the computer industry has a certain interest in maintaining these exponential trends. Still, it is amazing that such a tremendous miniaturization is possible at all.

Our understanding of what a computer can do is greatly unaffected by the enormous improvements resulting in ever faster computers. The foundations of theoretical computer science were laid in the 1930's by people like Kurt Gödel, Emil Post, Alonzo Church, and Alan Turing. The abstract mathematical models of computation that resulted from the work of these, and other, people well serve the purpose of being general models of computation that describe what, and what not can be done by a computer.

Even though computers have become much faster, they have been working according to the same principles, following the laws of classical physics. When the transistors accumulated on an integrated circuit reach the size of one, or a few atoms, the laws of classical physics no longer apply, and a further decrease in size requires to construct computers according to the laws of quantum mechanics. At least two conceivable ways of proceeding exist. One possibility is to accomplish the basic operations of a conventional computer within the framework of quantum mechanics. The other possibility is to build an entirely new computer based on quantum mechanical phenomena. At first it was tried to proceed according to the first possibility. However, it was soon realized that quantum phenomena allow for entirely new ways of computing. Not only can certain tasks, in the more narrow sense of computing as 'number crunching', be performed with an efficiency unparalleled by conventional computers. Quantum computers also allow for the detection of eavesdropping in key exchange, for secure message exchange, and for the generation of truly random numbers. The field of quantum information and computing is expanding rapidly as an area of research.

Another rather new research area that has the potential of developing into a new direction in computing is what is called DNA computing. With the revealing of the double helix structure of the DNA molecule, storing genetic information using four bases as molecular building blocks, where the order in which these bases occur encodes all information to build up an entire living animal, a molecular sized storage medium was discovered. Processing of the information encoded in a DNA molecule can be done by enzymes, and in 1997 it was demonstrated for the first time that DNA computing is a realistic possibility, by solving a small instance of the hamiltonian path problem, using DNA and enzymes only.

People differ grossly in opinion about the possible practical use of these new directions. Some believe that quantum computers will replace conventional computers within a few years, others doubt that they ever will be built. It is clear, however, that these new directions are interesting in itself, are leading to new insights, and may even redefine our

principal understanding of computers and computation.

This paper deals with some of the topics described above. The material that is discussed is based upon books and papers found in the references at the end. Certain aspects are treated in great detail in order to pinpoint essential ingredients, or fill in gaps in the source material. Other aspects are treated in a more narrative fashion, in order to emphasize the context in which the problem is discussed.

Chapter 2 touches upon the three traditionally central areas of the theory of computation , namely automata, computability, and complexity. These areas are all three involved in the fundamental question of what the capabilities of computers are. Automata serve as models of computation that are used in the other two areas. Computability distinguishes between problems that are solvable and problems that are not solvable by computation, whereas complexity classifies solvable problems in hard problems, for which no efficient algorithm exists or is known, and easier problems. This chapter also provides the background and fundamental concepts necessary for the remainder.

Chapter 3 starts with discussing the physical limitations that ultimately will cause Moore's law to break down. Next, the fundamental concept of the quantum bit, which is the quantum equivalent of the classical bit, is introduced. It is in detail explained how certain computational tasks can be accomplished with quantum logic gates, with an efficiency much higher than a conventional computer can achieve. A quantum logical circuit for performing the Fourier transform is presented. This also forms the basis for an efficient algorithm for factoring integers on a quantum computer.

The integer factoring problem plays an important role in cryptography which is the subject of chapter 4. Some of the basics of cryptography are discussed, especially public key cryptography. The RSA public key cryptographic system is discussed in relation to its dependence on the unproven assumption that no efficient factoring algorithm for factoring integers on a conventional computer, exists. Furthermore, a protocol, based on quantum mechanical principles, for establishing a secret key is discussed.

Finally, chapter 5 takes up another possibly promising direction in computing, namely DNA computing. The basic ideas are described together with a description of the first, experimental, DNA calculation.

Chapter 2

Theory of computation

2.1 Introduction

In most fields of science, problem resolution is based on the use of formal models. A formal model deals with mathematical objects that represent abstractions of the real entities to be modelled. Formal models basically require:

1. to formalize the problem, i.e. to choose a language that describes it

2. to solve the formal problem by means of the tools provided by the chosen formalism

Models play an essential role in computer science, computer scientists need models to represent a computer system at different levels of abstraction, for understanding it, analyzing it and proving properties, designing it, or even using it. We introduce two fundamental classes of computer science models, automata and grammars.

2.2 Finite automata

Finite automata can serve as models for a certain class of computations. To be more specific, finite automata can model computations that do not require the intermediate

storage of data because they lack the capability of storing data. Hence, finite automata may serve as the computational model for the kind of computations that can be performed on a computer with not only finite, but also extremely small amount of memory. As an example consider the following calculation. Given an input string consisting of the digits $\{0, 1, 2\}$, calculate the sum of the digits in the input string modulo 3 and accept the string if the result equals 0, otherwise reject. In figure 2.1 a graph representing the finite automaton performing this calculation is given. This graph can be thought of as an abstract machine that behaves in such a way as to actually perform the individual steps in a computation that lead to the required result. In the language of finite automata the graph in figure 2.1 is called a *state diagram*.

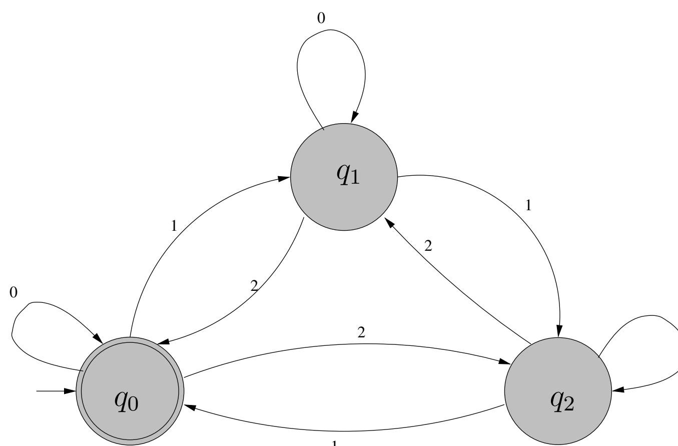


Figure 2.1: *A simple finite automaton, that checks if the sum of the input digits is a multiple of* 3*.*

The machine (finite automaton) works as follows. It consists of three states, labeled q_0, q_1, q_2, and arrows indicating transitions between these states. The digits labeling the arrows indicate a transition on input of that digit. There is at least one *accept state*, in this case q_0, indicated by a double circle. The *start state* is indicated by an arrow pointing at it from nowhere, here q_0. The machine starts in the start state and reads the symbols

from the input string, each of which yield a transition to a new state. If, at the end of input the machine is in an accept state it outputs *accept*, otherwise it outputs *reject*. For example on input of the string 12021 the finite automaton starts in state q_0, reads the digit 1 and changes to state q_1, next reads digit 2 and changes to state q_0, reads 0 and stays in q_0, reads 2 upon which it changes to state q_2, and, finally, reads 1 after which it changes to q_0 and returns *accept*.

In the example above the characters in the input string were restricted to the digits $0, 1, 2$. Such a set is generally called an alphabet. The transitions between the states are generally described by a *transition function* $\delta(r, s)$, indicating that state r changes to state δ on input s. Such a function can be represented by a table, as in table 2.1.

	0	1	2
q_0	q_0	q_1	q_2
q_1	q_1	q_2	q_0
q_2	q_2	q_0	q_1

Table 2.1: *The transition function $\delta(r, s)$ for the finite automaton in figure 2.1.*

A finite automaton can formally be defined as a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

1. Q is a finite set called the *states*
2. Σ is a finite set called the *alphabet*
3. $\delta : Q \times \Sigma \rightarrow Q$ is the *transition function*
4. $q_0 \in Q$ is the *start state*
5. $F \subseteq Q$ is the set of *accept states*

In many cases, but not always, a finite automaton can be depicted in a state diagram. If the machine contains a parameter, and thus in a way describes a whole class of machines it is impossible to depict this machine with a single state diagram.

Given a precise definition of a finite automaton it is possible to give a definition of computation, as follows. A finite automaton $M = (Q, \Sigma, \delta, q_0, F)$ is said to accept the string $w = w_1 w_2 \cdots w_n$ over the alphabet Σ if a sequence of states $r_0, r_1, \cdots r_n$ exists in Q, such that

$$\begin{array}{ll} 1. & r_0 = q_0 \\ 2. & \delta(r_i, w_{i+1}) = r_{i+1} \quad i = 0, 1, 2, \cdots n-1 \\ 3. & r_n \in F. \end{array} \tag{2.1}$$

The first condition requires the machine to start in the start state. Condition two requires the machine to change state from r_i to r_{i+1} upon reading symbol $w_{i+1} \in \Sigma$. The last condition expresses that the machine should be in an accept state when it has read the last symbol w_n and made the accompanying transition. The set A of all strings w accepted by machine M make up the language of the machine. Hence $A = \{w|M \text{ accepts } w\}$. Not all sets of strings are accepted by some finite automaton. However, if a set of strings is accepted by a finite automaton, that language is said to be *regular*.

The importance of the regular languages becomes clear if one considers operations on languages that produce new languages. Certain operations have the property that, if applied on regular languages, the result is again a regular language. These operations are called *regular operations*. In such a case it is said that the class of regular languages is *closed* under that operation. The following operations are regular: *union, concatenation, star*. They are defined as follows.

Union : $A \cup B = \{x|x \in A \vee x \in B\}$ The resulting language consists of all strings that are either in language A, or in language B, or in both.

Concatenation : $A \circ B = \{xy|x \in A \wedge y \in B\}$ The resulting language consists of all strings that result if a string from language A is concatenated with a string from

language B, in that order.

Star : $A^* = \{x_1 x_2 \cdots x_k | k \geq 0 \land x_i \in A\}$ This is a unary operation on language A, where an arbitrary number of strings are concatenated into a new string. Note that $k = 0$ also is allowed, hence the empty string, which is the result of the concatenation of zero strings and denoted ϵ, always is in A^*. Furthermore, A^* usually contains infinitely many elements, even if A only contains a finite number of elements, in contrast to the union and concatenation.

As an example consider the languages $A = \{ab, cd\}$ and $B = \{ef, gh\}$ over the alphabet $\Sigma = \{a, b, c, d, e, f, g, h\}$. Then

$$A \cup B = \{ab, cd, ef, gh\}$$

$$A \circ B = \{abef, abgh, cdef, cdgh\}$$

$$A^* = \{\epsilon, ab, cd, ef, gh, \cdots, efcd, efef, \cdots, ababab, ababcd, \cdots\}$$

The class of finite automata considered so far are usually called *deterministic finite automata*, abbreviated as DFA. They are deterministic because for every state and every valid input symbol the next state is given by the transition function, and hence is determined. There also exist more general finite automata like non-deterministic finite automata, abbreviated as NFA. We will slightly touch upon NFAs for the following reason.

Proving that the class of regular languages is closed under some operation is not so difficult for the union operation but is much more cumbersome for concatenation and star. The reason is that, for example in the case of concatenation a given string has to be accepted by a DFA if it can be broken into two sub-strings that each are accepted by some given DFA. The process of finding out how to partition a string can be done by trying out

all possibilities and only accepting those possibilities that yielded valid sub-strings. The process of finding valid sub-strings can be done with NFAs.

Non-deterministic finite automata differ from DFAs in the following two ways.

1. The transition function of a DFA specifies exactly one transition for each input symbol. For a NFA there can be zero, one ore more transitions *for each input symbol*. The behaviour of the machine is that it spits into multiple copies of itself if more than one transition exists for a given input symbol, and that it dies if there exists no transition for that input symbol. The table describing the transition function is now modified in that the resultant state is no longer a single state, but instead a set of states.

2. States may also split, and possibly change state, without having read any input. In the table for the transition function this is accomplished by an extra column labeled with the symbol ϵ. If the entry in this column is the empty set $\emptyset$ then the corresponding state cannot split.

Importantly, NFAs and DFAs turn out to be equivalent. That means that for each language that is accepted by an NFA there exists a DFA that also accepts that language. Because of this equivalence the proof of closure of the regular languages under the regular operations can be formulated in terms of NFAs.

The property of an NFA of being able to split in many copies that die if they do not end in an accept state allow for the construction of a simple NFA that is able to find valid sub-strings in an efficient way.

2.3 Regular expressions

A regular languages is a languages that is accepted by some DFA. One can construct new regular languages from existing ones with the help of regular operations. Since regular

operations take languages as arguments and also yield languages as result one can combine these operations to build more complicated expressions. Such expressions are similar to arithmetic expressions. One such similarity is that the value of such an expression is unique because of precedence rules. The star operation has the highest precedence, followed by concatenation, and finally union which has the lowest precedence. As usual one can change the meaning of a such an expression by inserting parentheses.

Some examples are in place. The expression $\{0, 1\}^*$ describes the set of all strings made up of zeros and ones. This expression can also be written as Σ^*, or as $(\{0\} \cup \{1\})^*$. Though formally incorrect the set symbols { and } are often left out if no confusion can arise. Hence the last form could be written as $(0 \cup 1)^*$. Also the symbol ϵ is often taken to stand for both the empty string and the set containing the empty string only, i.e., $\epsilon = \{\epsilon\}$. Similarly the concatenation symbol $\circ$ often is left out, like the multiplication symbol in mathematical expressions. Also note the following equalities:

- $A \circ \emptyset = \emptyset \circ A = \emptyset$ The empty set concatenated with any set yields the empty set.
- $\emptyset^* = \{\epsilon\}$ The result of the star operation is a set of strings, where each string is made up of the concatenation of a number of strings from the operand set. If the operand set is the empty set only the empty string results.

If R_1 and R_2 are regular expressions then, as we have seen, $R_1 \cup R_2$, $R_1 \circ R_2$, and R_1^* also are regular expressions. The simplest regular expressions do not contain any operators, examples are $\emptyset, \{\epsilon\}$, and $\{a\}$, where a is in the alphabet Σ. In summary the following inductive definition completely defines regular expressions.

- R is a regular expression if $R = \emptyset \vee R = \{\epsilon\} \vee R = \{a|a \in \Sigma\} \vee R = R_1 \cup R_2 \vee R = R_1 \circ R_2 \vee R = R_1^*$, where R_1 and R_2 are regular expressions.

Regular expressions applied on regular languages yield regular languages because of the closure property of the regular operations. This rises the question of whether there exist

other regular languages which are not the result of some regular expression. The answer to this question is no, all regular languages can be described by some regular expression. From the definition of a regular expression it is also clear that each regular expression yields a regular language. In other words, a language is regular if and only if some regular expression describes it. Finite automata and regular expressions are equivalent in their ability of describing regular languages. For each regular expression there exists a NFA that accepts the language described by the regular expression. For example the regular expression $\{ab, a\}^*$ is described by the NFa the state diagram of which is depicted in figure 2.2.

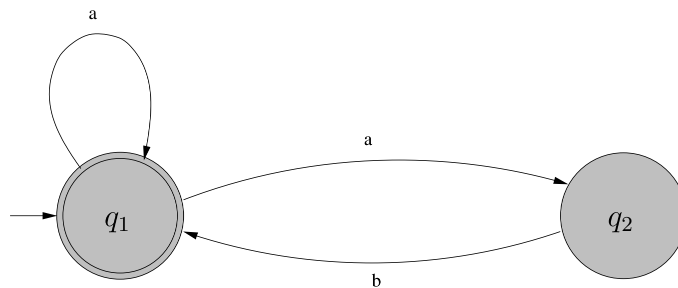


Figure 2.2: *The NFA corresponding to the regular expression* $\{ab, a\}^*$.

Some examples of languages that are not regular are

1. $A = \{0^n 1^n | n \geq 0\}$
2. $B = \{1^{n^2} | n \geq 0\}$
3. $\{w | w$ contains an equal number of 0s and 1s$\}$

These languages have the common property that they consist of strings with very special properties. It is not possible to construct a finite automaton that accepts only strings with these properties and no others. In other words a language that is regular and contains all

the strings from the first example above must necessarily also contain other strings, which can be proven by the pumping lemma.

2.4 The pumping lemma

All infinite regular languages share a certain property known as the pumping lemma [32], which says the following.

Pumping lemma If A is a regular language then there exists a number $n > 0$, called the pumping length, such that if s is a string in A with $|s| \geq n$ then s may be divided into three pieces x, y, z with $s = xyz$, where

1. $|y| > 0$
2. $|xy| \leq n$

and where

3. For all $k \geq 0, xy^k z \in A$

The pumping lemma specifies a necessary condition for regular languages. Hence it can be used to prove that certain languages are not regular, but it can never be used to prove that a language is regular. Note that the pumping lemma does not say anything about the language A being finite or infinite. The reason is that the pumping lemma is trivially true for finite languages. For a finite language one chooses n larger then the length of the largest string in A in which case the lemma also is satisfied.

The pumping lemmas is a consequence of the fact that a DFA with a finite number of states is able to accept strings that are much longer than the number of states in the DFA. As a consequence the DFA must *loop* when parsing such a string. If the substring that was parsed when executing this loop is repeated more then once, i.e. the loop is executed more

then once, then that string will also be accepted. In this way an infinite series of strings that all are part of the language is generated. This is the basic idea of the pumping lemma which also forms the basis for the proof, which we will not give.

Making use of this property it is however not so difficult to understand why the language in example 2 above is not regular. The language consists of a series of strings, the lengths of which form the quadratic series $\{0, 1, 4, 9, 16, \cdots\}$. Strings that are formed by repeating a substring once, twice, thrice, etc will form a sequence that grows linearly. Since no subset of a quadratic growing series grows linearly the language cannot be regular.

Also language $A = \{0^n1^n|n \geq 0\}$ from example 1 can be proven to be not regular with the pumping lemma. Assume that the pumping length of A is p. Then according to the pumping lemma string $s = 0^p1^p$, which obviously is in A, can be partitioned as $s = xyz$ such that $xyyz$ also belongs to A. However, no matter how one chooses y, either as a sequence of zeros, or as a sequence of ones, or as a mixture, if xyz matches the pattern 0^n1^n then $xyyz$ will not match, hence A is not regular.

2.5 Pushdown automata

Deterministic finite automata are limited in their capability to to serve as a general model of computation because there exist strings that these machines cannot deal with, as shown above. In the case of the language $A = \{0^n1^n|n \geq 0\}$, the reason is that the a DFA cannot in any way remember how may zeros it has read. Consequently it cannot test whether or not the number of ones equals the number of zeros.

Pushdown automata are generalizations of NFAs. Basically a pushdown automaton (PDA) is a NFA complemented with a stack. Apart from the input alphabet $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$ there is a stack alphabet $\Gamma_\epsilon = \Gamma \cup \{\epsilon\}$, which may be identical to the input alphabet. Both alphabets are complemented with the empty string in order to allow for non-determinism. The situation is depicted in figure 2.3.

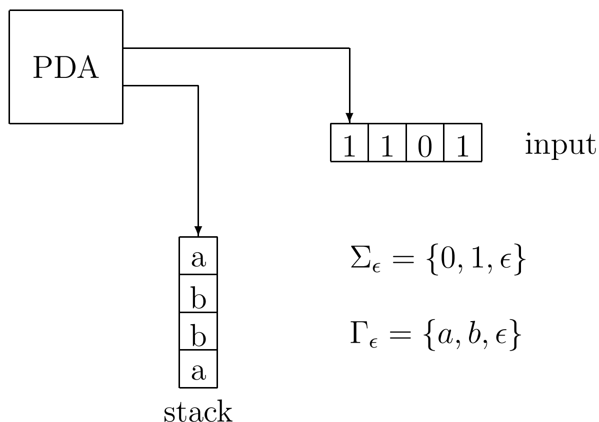


Figure 2.3: *Pushdown automaton with input and stack.*

The introduction of a a stack primarily affects the transition function. Previously the transition function specified a new state depending on the current state and the latest character read from input. Now the new state also depends on the character on top of the stack. Moreover, the stack will be modified in a way that also depends on the current state , the input character and the character on top of the stack. The table specifying the transition function becomes more complicated and we will discuss its appearance with the aid of an example, namely a PDA for the language $A = \{0^n 1^n | n \geq 0\}$.

As pointed out before, a NFA has no way to remember the number of characters read from input so far, at least not for strings of arbitrary length. For a language A' consisting of all strings in A up to a certain length it would be possible to construct a NFA that accepts the language A'. But that is not possible for A. A PDA can, however, put all the characters it reads on the stack meaning that it has an implicit count of the number of characters read so far.

Thus we can construct a PDA for language A by letting it reading 0s, which are all put on the stack, until a 1 is read after which the stack is popped, which is subsequently done for each next input character, if it is a 1. If at the end of input the stack is empty too then the string is part of A. The PDA doing just this is specified in table 2.2.

input	0			1			ϵ		
stack	0	\$	ϵ	0	\$	ϵ	0	\$	ϵ
q_1									$\{(q_2, \$)\}$
q_2			$\{(q_2, 0)\}$	$\{(q_3, \epsilon)\}$					
q_3				$\{(q_3, \epsilon)\}$					
q_4								$\{(q_4, \epsilon)\}$	

Table 2.2: *Specification of the transition function for a PDA accepting* $A = \{0^n 1^n | n \geq 0\}$*. Note: emty entries mean* $\emptyset$*.*

Testing for an empty stack can be accomplished by starting with putting a special symbol, \$ in this case, on the stack. Encountering of the \$ sign then indicates empty stack. otherwise only 0s have to be stored on the stack, hence $\Gamma = \{0, \$\}$, and $\Sigma = \{0, 1\}$. The entries in the table correspond to pairs (r, s), where r is the new state and s the value that replaces the value on top of the stack. Both the new and the old value on top of stack may be ϵ, indicating the omission of pushing a new value and popping the old value, respectively.

The state diagram corresponding to table 2.2 is given in figure 2.4. The notation $a, s \rightarrow t$, accompanying the transitions, means that upon input a the symbol s on top of the stack is replaced by symbol t. If $a = \epsilon$ the corresponding transition occurs without reading any input. If s or t equals ϵ this indicates the omission of popping or pushing, respectively, as before.

For completeness we also give the formal definition of a PDA. This merely differs from the definition of a DFA in the addition of the stack and accompanying modification of the

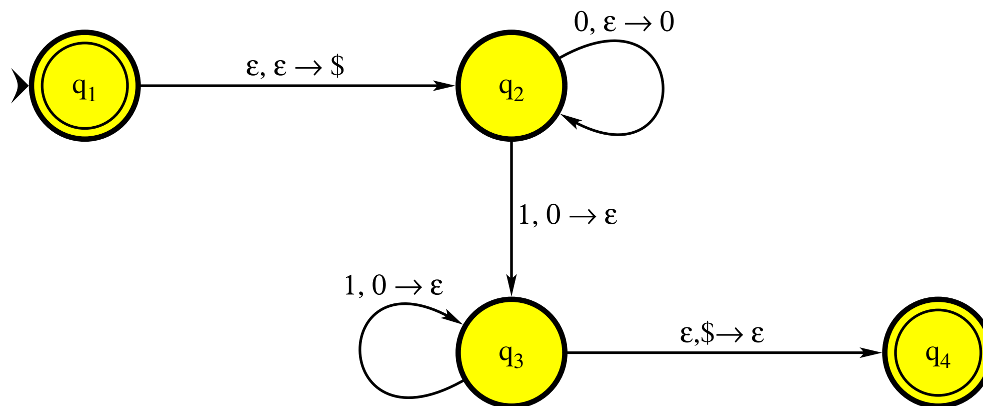


Figure 2.4: *State diagram of the PDA accepting the language* $A = \{0^n1^n|n \geq 0\}$.

transition function. A pushdown automaton is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are finite sets, and

1. Q is the set of *states*
2. Σ is the input alphabet
3. Γ is the stack alphabet
4. $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \rightarrow Q$ is the *transition function*
5. $q_0 \in Q$ is the *start state*
6. $F \subseteq Q$ is the set of *accept states*

2.6 Context free languages

A non-deterministic finite automaton is equivalent to a pushdown automaton without a stack. Thus, PDAs accept all languages accepted by a NFA, i.e. PDAs accept all regular languages. PDAs, however also accept certain non-regular languages, an example of which has been presented above. It has been shown that any language accepted by a PDA can

be generated by a context free grammar. The set of strings generated by a certain context free grammar (CFG) is called a context free language (CFL). Hence the class of languages accepted by PDAs is the class of context free languages.

A CFG is a set of rules where each rule specifies how a variable can be replaced by strings consisting of variables and other symbols called terminals. One variable is the start variable, by convention this variable occurs on the left hand side of the first rule. The language $A = \{0^n 1^n | n \geq 0\}$ is generated by the CFG

$$S \rightarrow 0S1 \tag{2.2}$$

$$S \rightarrow \epsilon. \tag{2.3}$$

This grammar contains only one variable, S, and the alphabet of the strings being defined is $\{0, 1\}$. The right hand sides of the substitution rules are also called *productions*. The generation of a certain string requires a sequence of substitutions that together form a *derivation*.

The derivation of the string 000111 is

$$S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 000S111 \Rightarrow 000111. \tag{2.4}$$

Note that the arrows used in a derivation are different from those in the rules that define the grammar. A derivation can equivalently be represented in the form of a parse tree. For the above string the parse tree looks as depicted in figure 2.5.

When performing a derivation there is a degree of redundancy in which variable to replace and which rule to choose for that variable. The first redundancy can be dealt with by doing derivations always in the same way. To be precise, one replaces the leftmost occurring variable at each step in the derivation. The second redundancy, choice of rule, will usually yield different strings. If not, there are two different leftmost derivations

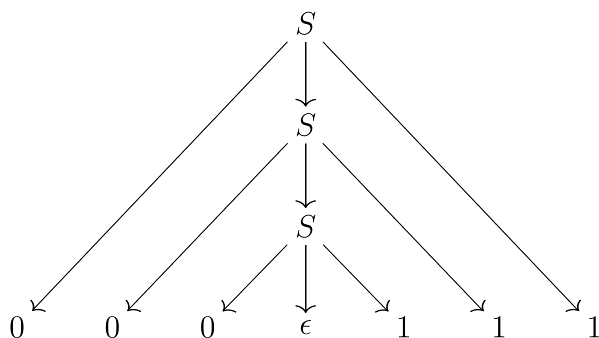


Figure 2.5: *Parse tree for the string* 000111.

yielding the same string in which case the grammar is ambiguous. In that case there also are two different parse trees for one and the same string.

Summarizing, a CFG consists of a 4-tuple V, Σ, R, S, where V, Σ, and R are finite sets, and where

1. V contains the variables.

2. Σ, *which is disjoint from* V, contains the terminals.

3. R contains the rules, which are of the form

$$< \texttt{variable} > \rightarrow < \texttt{variable(s) and/or terminal(s)} >$$

4. $S \in V$ is the start variable.

2.7 Equivalence of PDAs and CFGs

The fact that push down automata are equivalent to context free grammar is an important theorem of computer science. More precisely the theorem states that *A language is context free if and only if some push down automaton recognizes it.* In one direction the theorem states that if a language L is context free, then there exists a push down automaton

that recognizes language L. This direction of the theorem can also be proven by, for a general CFG, showing in detail how a PDA that recognizes the language of the CFG can be constructed.

A PDA recognizing the language of a CFG works as follows. As before it starts by writing some special symbol, i.e. a $ sign, on the stack in order to be able to recognize an empty stack. Next it writes the start variable on the stack. Now the topmost symbol on the stack is a variable and the next step is to non deterministically split into a number of copies of itself, one for each rule in the grammar that has the start symbol on the left hand side. Each of these copies writes the substitution symbols on the right hand side of the rule on the stack *in reverse order.*

Now the first symbol of the right hand side of the rule is on top of the stack. If this symbol is a variable the process of splitting and writing the rule on the stack is repeated. Otherwise the symbol on top of the stack is a terminal. This symbol consequently is the first symbol in the string generated by the grammar. The PDA has as input one of the strings of the language of the CFG. By letting the PDA non deterministically generating all the strings of the grammar, one of the copies of the PDA will generate the input string. Each time the symbol on top of stack is not a variable this matches the next input symbol. The symbol is then popped from the stack and the PDA reads the next input symbol. Again the stack is examined, and if a variable this is substituted for and the process continues in the same manner.

In figure 2.6 the the state diagram of a PDA that accepts all strings generated by the CFG given in grammar 2.2 is depicted. Starting at q_0 the $ and S symbols are written on the stack after which the PDA is in state q_2. From here there are two loops back two q_2, corresponding to the two rules of the grammar. Note that the loop $q_2 \rightarrow q_3 \rightarrow q_4 \rightarrow q_2$ pushes the symbols of the first rule in the grammar on the stack *starting at the end of the substitution*. The other loop, directly back onto q_2, corresponds to the emty replacement rule and also states the transitions that occur when the symbol on top of the stack is a

terminal.

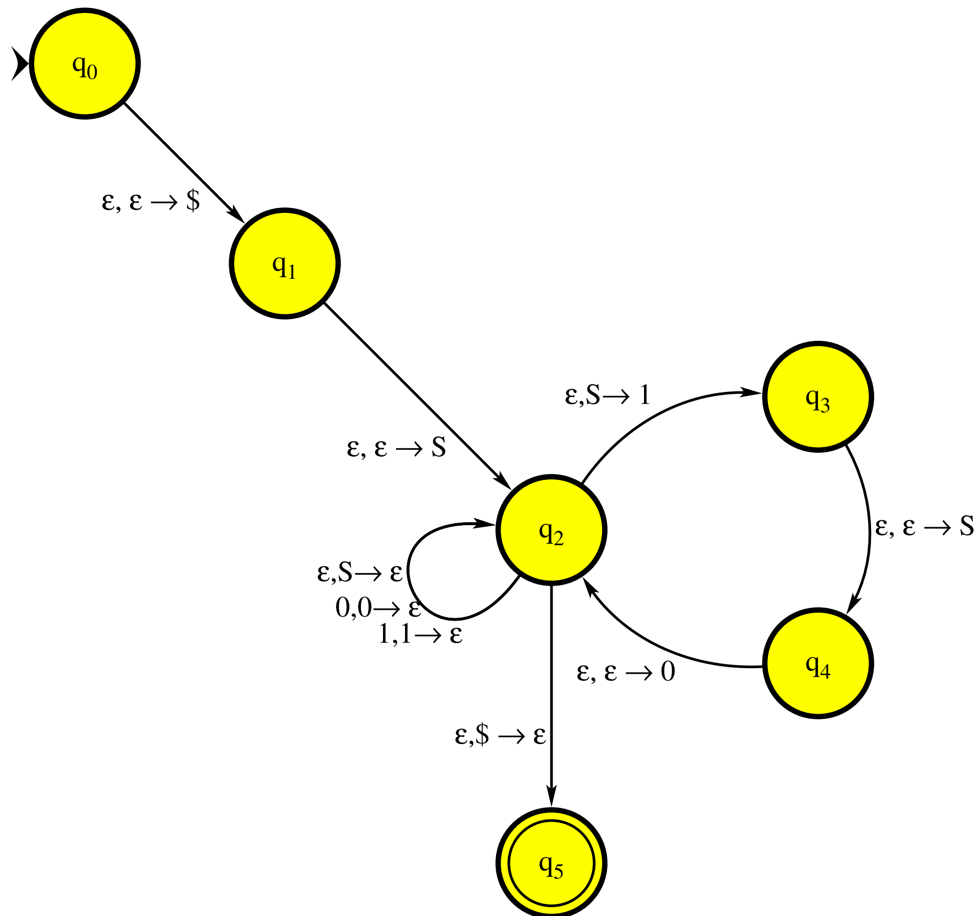


Figure 2.6: *State diagram of a PDA that accepts all strings generated by the CFG given in 2.2*

The PDA thus has the rules of the grammar built in, and more or less executes these rules which corresponds to a derivation of a string of the grammar.

2.8 The pumping lemma for context free languages

Also for context free languages there exists a pumping lemma, as is the case for regular languages. The pumping lemma states the following.

If A is a context free language then there exists a number $n > 0$, called the pumping length, such that if s is a string in A with $|s| \geq n$ then s may be divided into five pieces u, v, x, y, z with $s = uvxyz$, where

1. $|vy| > 0$
2. $|vxy| \leq n$

and where

3. For all $k \geq 0, uv^kxy^kz \in A$

It is immediately clear that upon taking $u = v = \epsilon$ the above pumping lemma reduces to the pumping lemma for regular languages, as required, because CFGs are equivalent with PDAs, which in turn are generalizations od DFAs. Also here the pumping lemma originates in the fact that automata have a finite number of states but are nevertheless capable of accepting languages containing strings of arbitrary lengths, hence also strings of lengths larger than the number of states of the PDA. That means, as before, that the PDA has at least one state that it assumes more than once, when dealing with strings of length larger than the number of states. The state that is assumes twice, together with the sequence of stats that it assumed in between, can be thought of as a loop. Then all strings that are generated by executing this loop any number of times will also be accepted by the PDA and hence be part of the language.

Because of the equivalence of PDAs and CFGs the pumping lemma can also be understood in terms of CFGs. The important observation again is that CFGs consist of *finite* sets of variables, terminals, and rules. Because strings of arbitrary length have a derivation base on a finite number of rules, the derivation of such a string must is obtained by applying at least one rule more than once. By repeating the sequence of steps that occurred between the repeated applications of that rule one obtains another string that

obviously also is part of the language. This sequence can be repeated any number of times, generating longer and longer strings that are all part of the language.

The above described process of generating new string in the language can be made visual by considering the parse tree, as is done in figure 2.7. Here R is the variable in the rule that was applied at least twice. An extra repetition of the intermediate steps is equivalent to replacing the sub-tree under the lowest R with the larger sub-tree under the highest R.

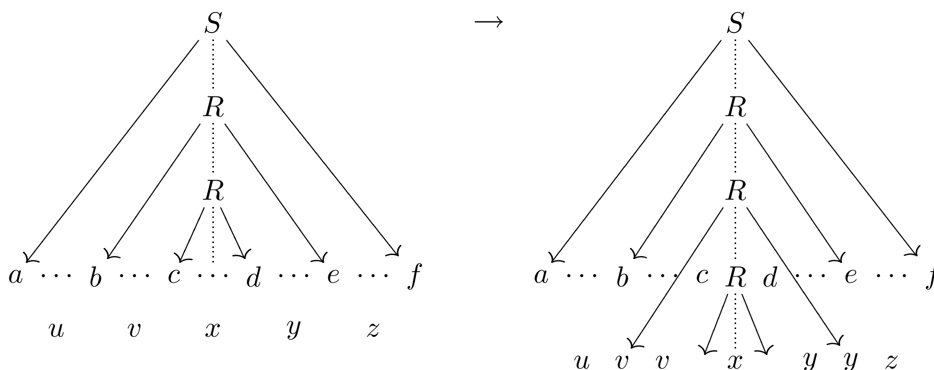


Figure 2.7: *Effect of pumping lemma on the parse tree.*

In the case of the non-regular but context free language $A = \{0^n 1^n | n \geq 0\}$ the shortest string that possibly can satisfy the conditions of the pumping lemma is the string 01. This string can be partitioned as $u = x = z = \epsilon$, $v = 0$, and $u = 1$. This is a rather special case where pumping this string generates the whole language.

Because the pumping lemma specifies a necessary condition for a CFL its main benefit is proving that certain languages are *not* context free. Examples of non context free languages are

1. the language consisting of strings of 1s of length a square integer: $B = \{1^{n^2} | n \geq 0\}$.
2. the language consisting of strings of 0s of length a power of 2: $C = \{0^{2^n} | n \geq 0\}$.

The pumping lemma generates series of strings the lengths of which increase linearly whereas the above languages generate strings that the lengths of which increase quadrati-

cally and exponentially, respectively. As a consequence the pumping lemma will generate strings that cannot be in the language from which it follows that these languages are not context free.

2.9 Turing machines

Push down automata are generalizations of deterministic finite automata and are consequently more powerful. However, since PDAs do not accept non-context free languages the concept of a push down automaton cannot serve as a general model of computation. Examples of languages that are not accepted by PDAs have been presented above.

In 1936 a much more general model of computation was presented by Alan Turing. The model, which is known under the name of a Turing machine, consists of a control unit, a tape head, and a tape of infinite length, see figure 2.8. The tape contains symbols from an alphabet Γ, which can be read and written by the tape-head. The tape-head can move in both directions over the tape, one symbol at a time, read the symbol and possibly overwrite it with a new symbol from the alphabet Γ.

The control unit can be in any of a finite number of states. When a symbol is read from the tape the control unit changes state, where the new state is a function of the newly read symbol and the current state. Initially the tape contains a string of symbols from the alphabet Σ, which is identical to Γ, but without the blank symbol. The input string fills a finite number of positions, starting on the left, and only, boundary of the tape with symbols from the alphabet Σ, while the rest of the tape is filled with blanks. The blank symbol will be denoted $\sqcup$, if its occurrence has to be emphasized.

The Turing machine continues processing until it reaches one of the special states *accept* or *reject*, or may continue processing forever. There exists variants of Turing machines that differ from the one described above. These differences may concern a tape that is unbounded in both directions, the use of more than one tape, the inclusion of non-

determinism, or the omission of *reject* states, in which case all non-*accept* states become *reject* states if the machine stops there. The important thing is that all these variants of Turing machines turn out to be equivalent in the sense that they all accept the same *class* of languages.

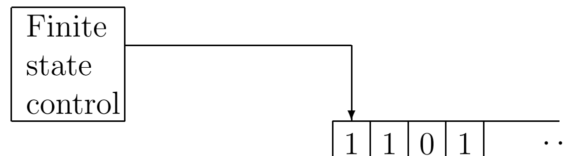


Figure 2.8: *Turing machine.*

In summary a Turing machine can be formally defined as a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, Q_{accept}, Q_{reject})$, where Q, Σ, and Γ are all finite sets, and

1. Q is the set of states.

2. Σ is the input alphabet, which may not contain the blank symbol $\sqcup$.

3. Γ is the tape alphabet, which is a superset of the input alphabet, $\Sigma \subset \Gamma$, and always contains the blank symbol $\sqcup$.

4. The transition function is $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$. The symbols L, and R indicate the direction in which the tape-head will move after having read a symbol from the tape.

5. There is one start state $q_0 \in Q$.

6. The accepts states $Q_{accept} \in Q$.

7. The reject states $Q_{reject} \in Q$.

A Turing machine can do everything a real computer can do, this is a consequence of the Church-Turing thesis which we will discuss later. It is for example possible to design

a Turing machine that recognizes the language $B = \{1^{2^n} | n \geq 0\}$. The machine uses the fact that a number that is a power of two can be divided by two with result always an even number until the result is the number 1. A number that not satisfies this rule is not a power of 2.

In figure 2.9 the state diagram of a Turing machine that recognizes the language $B = \{1^{2^n} | n \geq 0\}$ is depicted. A label of the form $a \rightarrow b, D$ on a transition arrow means that the symbol a on the tape is overwritten by the symbol b, after which the tape head is moved in the direction D, where D is one of L, R, standing for left and right, respectively. If the symbol b is left out the symbol a is left on the tape.

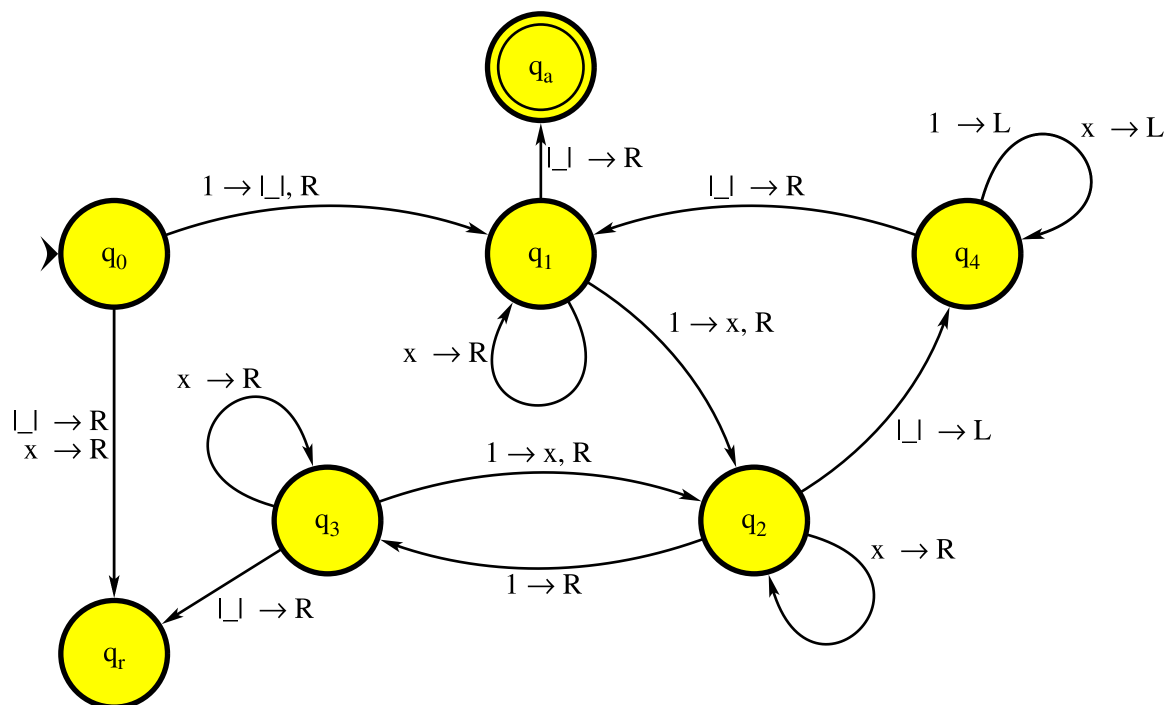


Figure 2.9: *State diagram of a Turing machine recognizing the language* $B = \{1^{2^n} | n \geq 0\}$

The machine starts by replacing the first 1 with a blank, in this way it can easily test for the beginning of the string when moving to the left. If the next symbol is a blank the machine changes to the accept state q_a, since 1 also is a power of 2. Otherwise it reads the next 1, replaces it by a x, and arrives in sate q_2. From here the machine starts reading

the 1s in pairs, crossing off every other 1, while hopping between the states q_2 and q_3. If the number of 1 is odd a $\sqcup$ will be encountered while in state q_3, after which the machine assumes the reject state q_r. At the end of the sweep the machine has to return to the beginning of the string, which is achieved through the state q_4, until the blank marking the start of the string is encountered and state q_1 is assumed again. If the string consists of xs only the string is accepted and q_a is assumed. Otherwise a new sweep is started, with crossing off every other 1.

2.10 The Church-Turing thesis

The Turing machine is a very general model of computation. It turns out that that everything that can be computed, by any means, can also be computed by a Turing machine. The implication of this is that problems that cannot be solved by a Turing machine can not be solved at all. Computers solve problems by executing what is usually called an algorithm. An algorithm usually is understood to be a set of rules that in detail describe the individual steps that will lead to a solution of the problem at hand. The question is what kind of problems can not be solved by a Turing machine. It is currently believed that for any problem for which one can formulate an algorithm in the sense of the above description, there exists a Turing machine that can execute that algorithm and hence solve the problem.

As a consequence Turing machines can be considered to be *equivalent* to algorithms. This leads to a conceptual framework for computation where a Turing machine *defines* an algorithm. Everything what can be expressed in the form of an algorithm as a set of rules can also be formulated as a Turing machine, which is called the Church-Turing thesis. This thesis, which also can be taken as a conjecture, basically says that the intuitive notion of an algorithm as a kind of recipe is equivalent to a Turing machine.

An implication of the Church Turing thesis is that what cannot be computed by a

Turing machine cannot be computed at all. In other words, for a problem for which no Turing machine exists that solves that problem there exists no algorithm that solves that problem. Such a problem is algorithmically unsolvable. Hence, of all models of computation considered so far, the Turing machine is the most general, and currently no more general model seems to be possible at all.

An example of a problem for which no algorithm exists that solves the problem is the general problem of software verification. More specific, it is not possible to design a program that, given a program and input to the latter program, tests whether the latter program produces the correct output. This can be proven by contradiction.

First a class of programs to be tested for correctness is chosen. We chose the programs P that print out the string `no` if P is given itself as input, and the string `yes` for any other input. Now the existence of a tester H is assumed that tests the correctness of P, i.e., tests whether or nor P prints out `no` if given itself as input. Since P is to be tested for correctness if given itself as input, H only needs P as input.

H is constructed such that it produces the string `yes` if P is correct, and the produces the string `no` if P is incorrect. Since H can test any program we can feed H itself into H.

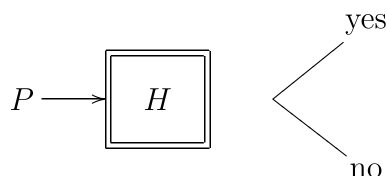


Figure 2.10: *Testing program correctness.*

Suppose that H now outputs `yes`, in that case H states that H is a program that will output the string `no` if it reads its own description as input. This is a contradiction because H outputs `yes` on assumption. If, on the other hand, it is assumed that H outputs `no` then H states about itself that it is a program that will *not* output `no` if provided with its own description as input. But it *did* output `no`, this is again in contradiction with the assumption. Since the assumption of the existence of H always leads to a contradiction

the conclusion is that H does not exist.

2.11 The halting problem

The non-existence of solutions in the form of an algorithm to certain problems is related to the way Turing machines work. Given a certain input, the machine may end up in an accept state, in a reject state, or simply never reach either an accept or a reject state, in which case the machine will not halt. The above proof of the non-existence of a Turing machine that tests the correctness of another Turing machine shows that a supposed correctness-tester will either produce output which differs from the required output, i.e., wrong output, or will never halt.

The problem of determining whether or not a given Turing machine M will halt on a given input string w is called the halting problem. This problem has no solution. For a more precise formulation somewhat more terminology is needed. The set of strings L that is *accepted* by a Turing machine M is called the language of M, and denoted as $L(M)$. If a string s is not accepted by a Turing machine M, then it is either rejected, or M never halts. Certain Turing machines halt on every input, i.e., either accept, or reject. Such a Turing machine is called a *decider*. If a language L is accepted by a decider then that Turing machine is said to *decide* language L, in which case L is called a decidable, or sometimes Turing-decidable, language. A decider *rejects* all strings that do not belong to the language. The proof of the undecidability of the halting problem is by contradiction, following the same kind of reasoning as used in the proof of the non-existence of an algorithm for software correctness verification, given above.

Each decidable language is accepted by a Turing machine, namely the decider that makes the language decidable. There are, however, languages that are accepted by a Turing machine, but which are not decidable. Hence, for such a language there exists no decider that decides that language. If the Turing machine that accepts such a language

is presented a string that does not belong to the language, it may reject or never halt. A language accepted by a Turing machine is called Turing recognizable, and is also called a *recursively enumerable language*, whereas a decidable language also is called a *recursive language*.

In order to determine whether or not a given problem has a solution or not the problem can be reformulated as a decision problem. That means that the reformulated problem is to find out whether or not a certain language is decidable. For example the problem of whether or not it is possible to determine whether or not a given DFA accepts a given string can be reformulated as follows. The DFA has a representation as a string D. This string, together with the supposed input string s forms a new string denoted as $< D, s >$. The original problem can now be stated as to determine whether or not the language

$$A_{DFA} = \{< D, s > |D \texttt{ accepts } s\} \tag{2.5}$$

is a decidable language. It can rather easily be proven that A_{DFA} *is* a decidable language. In other words, it is always possible to determine whether or not a string $< D, s >$ belongs to language A_{DFA}. If $< D, s > \notin A_{DFA}$ then D does not accept s, if $< D, s > \in A_{DFA}$ D does accept s.

Similarly it is alway possible to determine whether or not a given push down automaton accepts a given string. Formulated as a decision problem this means that, in notation equivalent to the above used notation, A_{CFG} is a decidable language. For Turing machines the situation is different. It is not always possible to determine whether or not a given Turing machine will accept a given string. The language

$$A_{TM} = \{< M, s > | \texttt{ Turing machine } M \texttt{ accepts } s\} \tag{2.6}$$

is not decidable. This means that there are strings for which no Turing machine will be able to determine whether or not the string belongs to A_{TM}. In other words, there exists,

according to the Church-Turing thesis, no algorithm that given a Turing machine and a string, will be able to determine whether or not the string will be accepted. That does not mean that that it always is impossible to determine whether or not a certain string will be accepted by a certain Turing machine. But there exists no algorithm that works for all strings and Turing machines, and an algorithm that only works in special cases is not an algorithm, especially not if it is not known in advance for which string the algorithm will work.

The fact that the language A_{TM} is not decidable is closely related to the halting problem referred to above. Also the halting problem can be formulated as a decision problem.

$$A_{HALT} = \{< M, s > \mid \texttt{Turing machine } M \texttt{ halts on input } s\} \tag{2.7}$$

It can be shown that if A_{HALT} is a decidable language than A_{TM} is a decidable language which is a contradiction, and therefore A_{HALT} is not decidable. The proof is straightforward, but will be omitted here. The proof of the undecidability of A_{TM} will also be omitted, but we will discuss some of the background of the proof. The idea is based upon the observation that the number of languages over a given alphabet Σ is larger than the number of strings that can be constructed from Σ. Because a Turing machine can be encoded into a string, this means that there are more languages than Turing machines. Hence there must exist languages for which no Turing machine exists that will accept that language. For such a language certainly no decider exists either. From a finite alphabet an infinite, though countable, number of strings can be constructed. However, the number of languages that can be constructed is uncountable, which is the reason why there are more languages than strings.

Both DFAs and PDAs will always halt, that follows immediately from the way these machines compute. The last step in the computation is performed when the last symbol is read from the input, see for example equation 2.1. Because the input string is finite the machine will always halt. As a consequence the languages these machines accept

are restricted to belong to certain classes of languages, regular languages, or context free languages, respectively. Examples of strings not accepted by these machines have been given. The number of steps in the computation of a Turing machine is not limited, as in the case of DFAs and PDAs, as a consequence the machine may never halt on certain inputs.

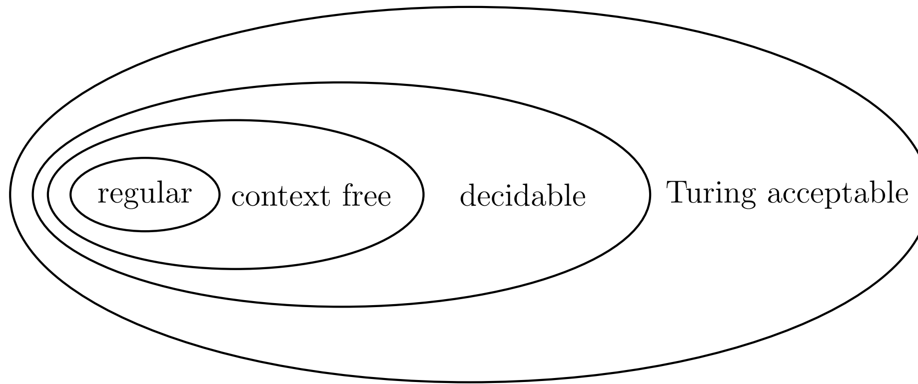


Figure 2.11: *Hierarchy of languages.*

2.12 Hilbert's tenth problem

It was through the work of Turing[1] and a number of other people, among them Alonzo Church, Kurt Gödel, and Emil Post, that the concept of an algorithm obtained a precise definition. Only with a precise definition it was possible to prove that certain problems could not be solved by an algorithm. As a matter of fact the very notion that certain problems might not have a solution was unknown before the work of the above mentioned people. A famous example is Hilbert's tenth problem. This is the tenth problem on a list of twenty-three problems that David Hilbert proposed as the most important, yet at the time (1900) unsolved problems.

The tenth problem was on polynomials, with integer coefficients, equated to zero and reads: (translation from the German) *To devise a process according to which it can be*

determined by a finite number of operations whether the equation is solvable in integer numbers. The formulation of the problem strongly suggests that Hilbert did not doubt the existence of such a process. Note especially that Hilbert required a *finite* number of operations for this process.

Even though the word *algorithm* was not used, the formulation of the problem, coincides very well with our intuitive notion of an algorithm as a recipe for performing a certain task. As such, an accurate definition of an algorithm is not necessary, if one can invent the process envisaged by Hilbert, then that would be the algorithm.

Hilbert's tenth problem has no solution, the sought algorithm does not exist. Hence, formulated as a decision problem, Hilbert's problem is not decidable. In other words, the set

$$D = \{p | p \text{ is a polynomial with integral coefficients and at least one integral root } \} \quad (2.8)$$

is not decidable.

Nevertheless, many polynomials with integer coefficients do have zeros at integer argument-values. But a *universal* solution to the problem does not exist. Especially, the set of single variable polynomials is decidable. Also the set D is Turing-recognizable.

The undecidability of the above set D was proven in 1970 by Yury Matiyasevich[2]. Actually Matiyasevich proved the last step required for the proof of the Davis conjecture, now also known as the DPRM theorem, after Davis, Putnam, Robinson, and Matiyasevic. The DPRM theorem says that every recursively enumerable set is Diophantine which as a consequence means that prime numbers are representable by a polynomial formula.

In 1976 such a representation was given by Jones, Sato, Wada, and Wiens[3], see equation 2.9. $P(a, b, c, ..., z)$ is a polynomial of the 25-th degree in the 26 variables $a, b, \cdots, z$, which assume positive integer values. The set of positive values of the polynomial is equal to the set of all prime numbers.

$$\begin{aligned}
P(a,b,c,...,z) \quad = (k+2)(1 \quad & -[wz+h+j-q]^2 \\
& -[(gk+2g+k+1)(h+j)+h-z]^2 \\
& -[2n+p+q+z-e]^2 \\
& -[16(k+1)^3(k+2)(n+1)^2+1-f^2]^2 \\
& -[e^3(e+2)(a+1)^2+1-o^2]^2 \\
& -[(a^2-1)y^2+1-x^2]^2 \\
& -[16r^2y^4(a^2-1)+1-u^2]^2 \\
& -[((a+u^2(u^2-a))^2-1)(n+4dy)^2+1-(x+cu)^2]^2 \\
& -[n+l+v-y]^2 \\
& -[(a^2-1)l^2+1-m^2]^2 \\
& -[ai+k+1-l-i]^2 \\
& -[p+l(a-n-1)+b(2an+2a-n^2-2n-2)-m]^2 \\
& -[q+y(a-p-1)+s(2ap+2a-p^2-2p-2)-x]^2 \\
& -[z+pl(a-p)+t(2ap-p^2-1)-pm]^2)
\end{aligned} \tag{2.9}$$

At first sight the formula seem to contradict the definition of prime numbers since the righthand side is a term consisting of two factors. The second factor however consist of a 1 minus a sum of squares, which equals 1 if all the squares are zero. Hence, if the variables $a, b, c, \cdots, z$ assume such values were all the squares are zero, and where the resulting value for P is non-negative, then this value for P is prime. So far, no solutions have been found.

2.13 Self-reference

Each Turing machine has a representation as a string over some alphabet. Moreover, Turing machines can write out and read in strings, and even simulate other Turing machines. A question with a non-obvious answer is whether or not a Turing machine is able to write out *its own description*. Everyday life-experience suggests that a machine that produces something is more complex than the produced object. From this one would conclude that a machine cannot print out its own description.

It actually is possible to construct a Turing machine that prints its own description, such a machine is depicted in figure 2.12. The machine is built up of two parts, A, and B, that execute one after the other, A handing over control to B.

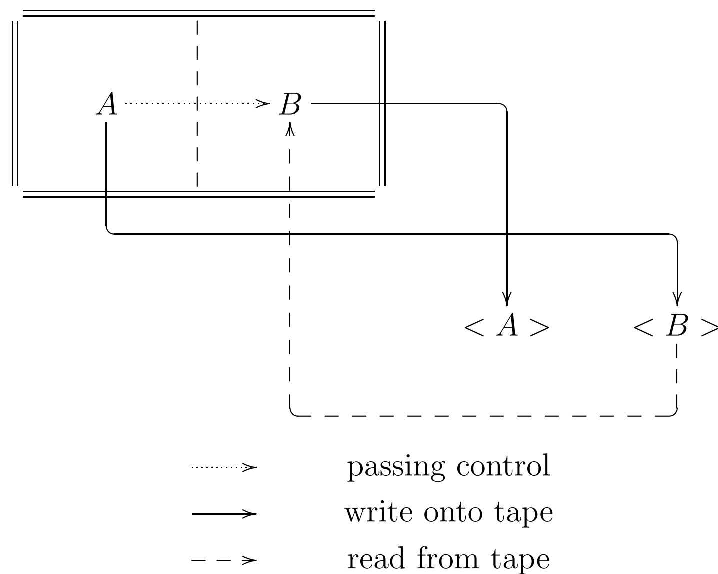


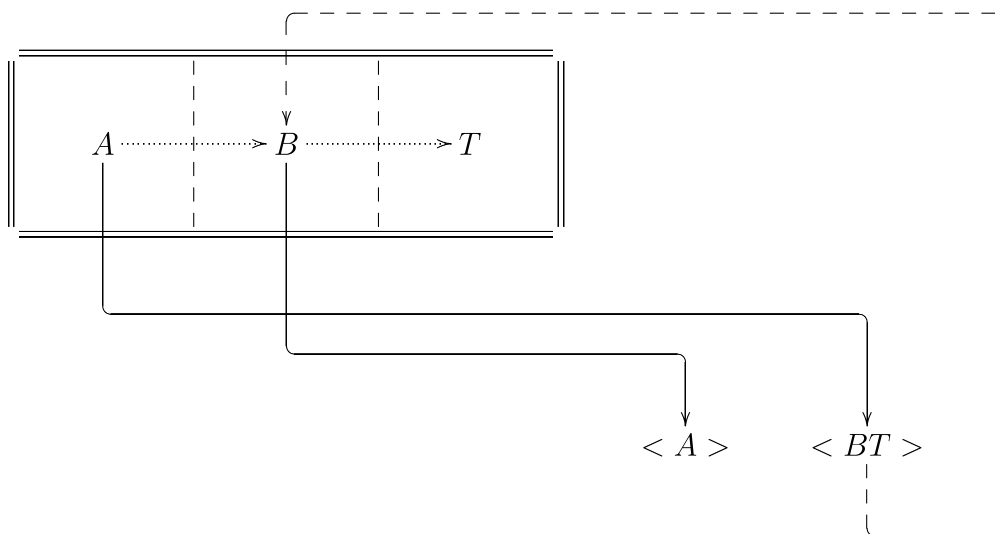
Figure 2.12: *Turing machine that writes out its own description*

The idea is that A prints out a description of B, after which B prints out a description of A. Part A has the description of B built in, and all that A does is printing out this string and then halt. From this it follows that B can not have the description of A built in, because B is not yet defined. However, the description of B is on the tape after A has

executed. Hence B can obtain its description $< B >$, and then *construct* A as the Turing machine that prints out the string $< B >$. Now B is defined as the Turing machine that on input of some string w first constructs a Turing machine P that neglects its input and next prints out this string, and second prints out the description $< P >$ of P. This latter description is the description of A, and $< P >$ should be printed in front of the description of B, in order to make a complete description of a Turing machine printing out its own description.

The machine in figure 2.12 only prints out its own description and then halts. It is possible to generalize the machine to one that prints out its own description and then continues. It can for example read in its own description and compute with it.

Actually every Turing machine can be complemented in such a way that it can write out and then read in its own description. In other words, for any computational task it is possible to construct a Turing machine that can obtain its own description.



For the meaning of the arrows see figure 2.12.

Figure 2.13: *Turing machine that can obtain its own description*

The general design of such a machine is given in figure 2.13, which can almost, but not quite, be obtained from figure 2.12 by replacing B by BT. The machine now consists of

three parts, A, B, and T, that execute in order. As before, B prints out a description of A, but now A prints out a description of BT. The machine B now hands over control to T instead of halting, otherwise it is the same as before, even though it now receives $< BT >$ as input instead of $< B >$. The part T can be whatever one pleases.

2.14 Complexity classes

Problems that are undecidable have no algorithmic solution. Decidable problems do have an algorithmic solution and, at least *in principle*, this solution can be computed. Often a problem contains a parameter such that different values of the parameter constitute different problems. Even if the problem has a solution for all values of the parameter, in practice it is only possible to obtain the solution for a very limited number of values of the parameter because the computational resources required for obtaining the solution are not available, or plainly non-existent.

As an example of such a problem consider the *travelling salesman problem* (TSP) which is the following. A salesman is supposed to visit a number, say N, of cities, and wants the shortest way such that each city is visited only once. Normally, many of the N cities have roads directly to at least some other cities, otherwise there is only one possibility which then also is the solution. Here, N is the parameter classifying the individual problems.

There is no efficient method known for solving the TSP. The only way is to compute the lengths of all routes and then picking the smallest. This method of testing all possibilities is called brute force. For small values of N this can easily be done. However, the number of possible routes is an exponential function of the number of cities. That means that the computational time to compute the solution also is an exponential function of N. Hence, for most values of N obtaining the solution is infeasible. Such a problem is generally called *intractable*.

Problems can be classified according to the asymptotic behaviour of the computational

time as function of the size of the input. This leads to a number of so called complexity classes, some of which are given in table 2.3.

Class	Description	Example
P	Polynomial time	CFL
NP	Non-deterministic polynomial time	Factoring integers
NP-complete	Subset of NP	Scheduling
ZPP	Polynomial time, PTM	

Table 2.3: *Some complexity classes.*

The class P consists of all problems that are solvable in polynomial time, these problems are considered tractable. An example is context free languages, which are polynomial time decidable.

A non-deterministic Turing machine may, in analogy with a non-deterministic finite automaton, at any point in the computation proceed according to several possibilities. The input will be accepted if some branch of the computation leads to the accept state. The class NP consists of all problems that are solvable in polynomial time on a non-deterministic Turing machine. The acronym NP stands for non-deterministic polynomial time. An equivalent deterministic Turing machine can be constructed that will need an exponential amount of time to solve the same problem.

There exist many problems that have the peculiar property that no polynomial time algorithm for obtaining the solution is known, but given a solution then the proposed solution can be checked for correctness in polynomial time. An example is the factoring of integers into prime numbers. No polynomial time algorithm is know for this problem, despite huge efforts. However, given an integer and its factors, then one can easily check the correctness by simply multiplying the factors, which can be done in polynomial time.

Hence, the factoring problem can be solved in polynomial time on a non-deterministic Turing machine by simply letting it check all possibilities. This basically is a parallel process where only the process that guessed the right factor survives. However, there is

the possibility that an efficient, i.e., polynomial time algorithm for the factoring problem will be found, since the non-existence of such an algorithm has not been proven. Since this applies to all problems in NP, there is the possibility that P and NP are equal.

The class ZPP contains all problems that can be solved in (average) polynomial time on a probability Turing machine. Such a machine can be seen as a kind of non-deterministic Turing machine were the next move is determined by a random process. Hence, the machine does not execute in parallel, but on different runs the machine will go through different sequences of states, depending on the outcome of a random generator. Certain problems are more efficiently solved on a probabilistic Turing machine than on a deterministic Turing machine, this will not be discussed here. In figure 2.14 the suspected relation between the classes P, ZPP, and NP is depicted.

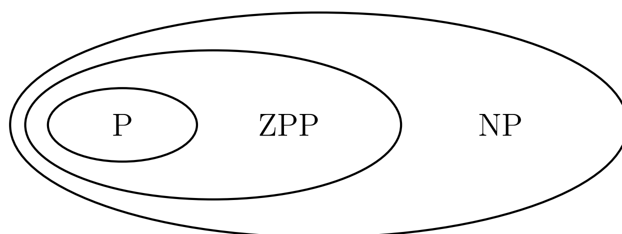


Figure 2.14: *Probable relation between complexity classes.*

The class of problems in NP also contains a subset of problems, called NP-complete problems, which have the property that *all* problems in NP can be reduced to an NP-complete problem. Moreover, the reduction-algorithm itself is in P. The existence of NP-complete problems plays an important role in the P versus NP question. It suffices to find an efficient solution to just one NP-complete problem in order to prove the equality of P and NP. Similarly, if one NP-complete problem is *proven* to not have an efficient solution, then P and NP will be different. The assumed situation is depicted in figure 2.15.

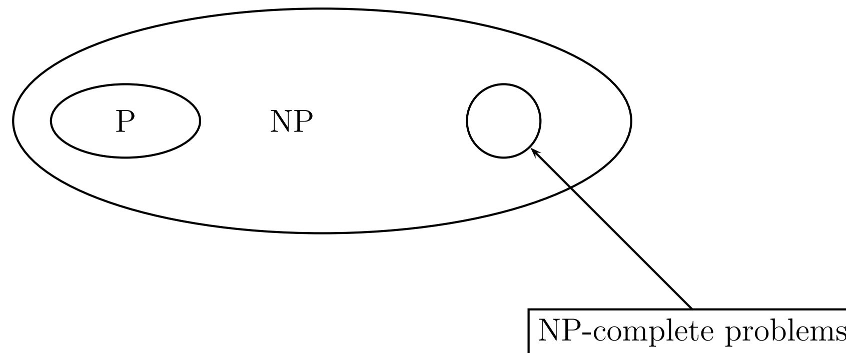


Figure 2.15: *Probable structure of the P and NP complexity classes.*

It is currently widely assumed that P and NP are different and that NP-complete problems do not have an efficient solution. Consequently, if a problem is proven to be NP-complete this is taken as strong evidence that the problem cannot be solved in polynomial time.

2.15 Summary and outlook

In this chapter a number of rather well known aspects of theoretical computer science have been discussed. It has been shown that there exist problems for which no algorithmic solution exist. Even if an algorithm for solving a given problem exists, it may be practically impossible to solve the problem because the number of computational steps is too large. Factoring integers of more than 200 digits is such a problem, the computational time required for factoring, using the fastest known algorithm, is an exponential function of the number of digits.

In the next chapter we will discuss an algorithm for a perceived quantum computer that

is able to factor integers with polynomial time complexity. This is one of a few currently known algorithms that achieve a certain task on a quantum computer with an efficiency higher than can be achieved on a conventional computer.

The integer factoring problem is the basis for the RSA public key cryptographic system. Although quantum technology thus is a threat against the RSA system, quantum technology also provides an alternative solution, this will be discussed in chapter 4.

Another possible future way of tackling NP-complete problems is taken up in chapter 5. Here the complexity of the algorithm is unchanged, but the computational time is reduced enormously through what in principle is massive parallelism.

Chapter 3

Quantum computing

3.1 Moore's law

Modern computers contain a central processing unit (CPU) that consists of a single silicon chip. This has been made possible by the invention of the transistor in 1947. The first generation of computers had vacuum tubes, these were replaced by transistors in the second generation of computers during the 1960s. As a consequence computers became smaller, more reliable and consumed less power. Next the integrated circuits appeared, which had more than one logic gate on a single chip. The integrated circuit has evolved, containing more and more logic gates, nowadays many millions.

The increase of the number of transistors on an integrated circuit over the years is given in table 3.1, together with the minimum feature size used in the production process, and the names of certain processors.

Gordon Moore, one of the founders of the Intel company, noticed that the number of transistors on a single chip approximately doubled every 24 months. This quantitative form of the increase of the number of transistors has come to be known as Moore's law. Figure 3.1 presents Moore's law in a plot, where the vertical axis shows the base 2 logarithm of the transistor count.

Year	Processor	Count	Size (μm)
1971	4004	2300	10
1972	8008	3500	10
1974	8080	6000	6
1976	8085	6500	3
1978	8086	29000	3
1982	80286	134000	1.5
1985	80386	275000	1.5
1989	Intel 486	1.2×10^6	1
1993	pentium	3.1×10^6	0.8
1995		5.5×10^6	
1997	pentium II	7.5×10^6	0.35
1999	pentium III	28.0×10^6	0.180
2001		44.0×10^6	
2003		95.2×10^6	
2005		190.0×10^6	

Source: Science and engineering indicators 2002, `http://www.nsf.gov/sbe/srs/seind02/`, table 8-1
Values for the years 2003, and 2005 are projected.

Table 3.1: *Transistor count per chip*

The open circles correspond to the values from table 3.1, the full line is least square fit of these points to a straight line. It is impressive to see how well the computer industry has succeeded in sustaining the trend in miniaturization starting from 1971. The least square fit resulted in the figure of 25 months for a doubling of the transistor count.

The increase of the transistor count affects both the memory capacity and the speed, i.e. the clock frequency of the processor. The reason is that the size of chips has stayed more ore less constant. Hence the individual transistors are packed more tightly. Thus the distance between components has decreased, which is a prerequisite for faster communication because the speed of the signals that are exchanged is a constant, namely the speed of light. Hence the clock frequency at which CPUs operate has increased exponentially

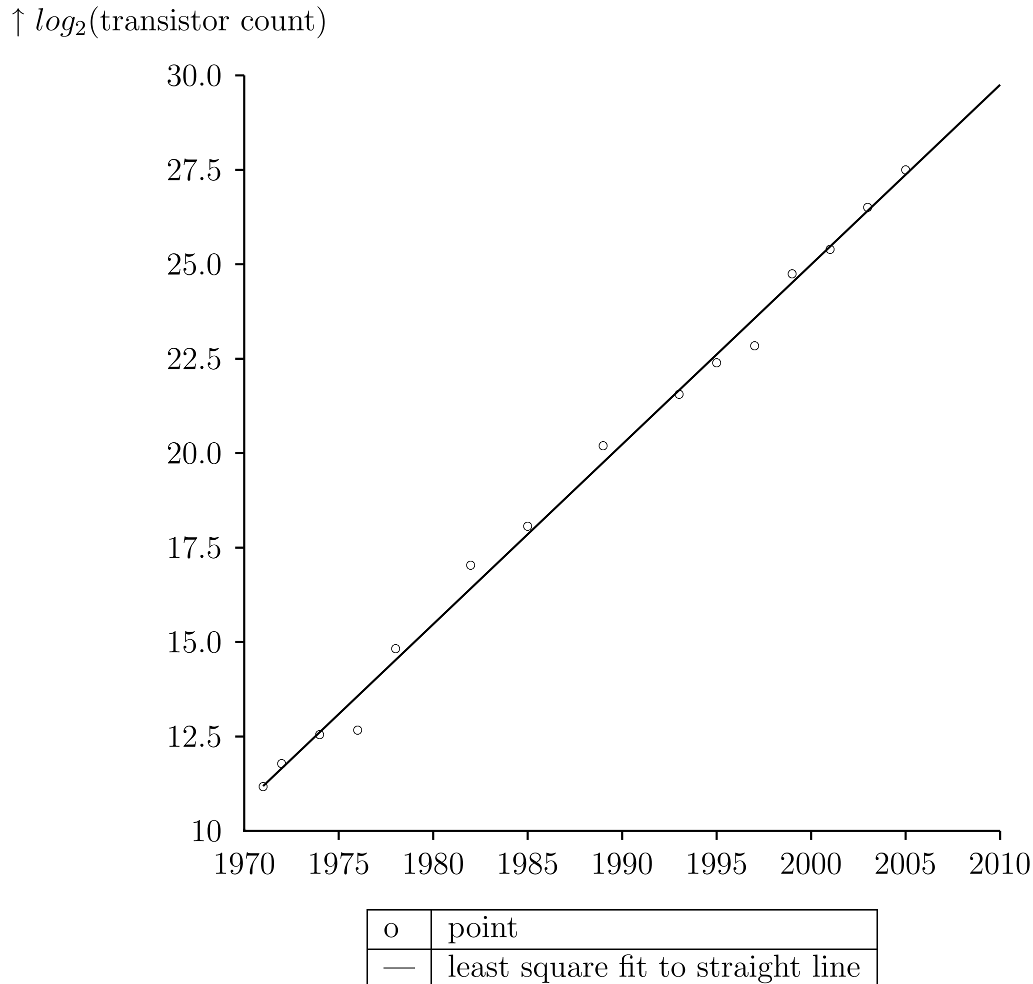


Figure 3.1: *Moore's law*

too, as a direct consequence of the exponential increase of the number of transistors on an individual chip.

The cost of fabrication of integrated circuits has been more or less constant. As a consequence, the average consumer computer doubled its performance, both in speed and memory capacity, at the same cost.

The question is how long the exponential increase of speed and capacity can continue. There are two important aspects to be considered in connection with this question. These are

1. The amount of energy dissipated per logical operation.
2. The number of atoms used to store one bit of information.

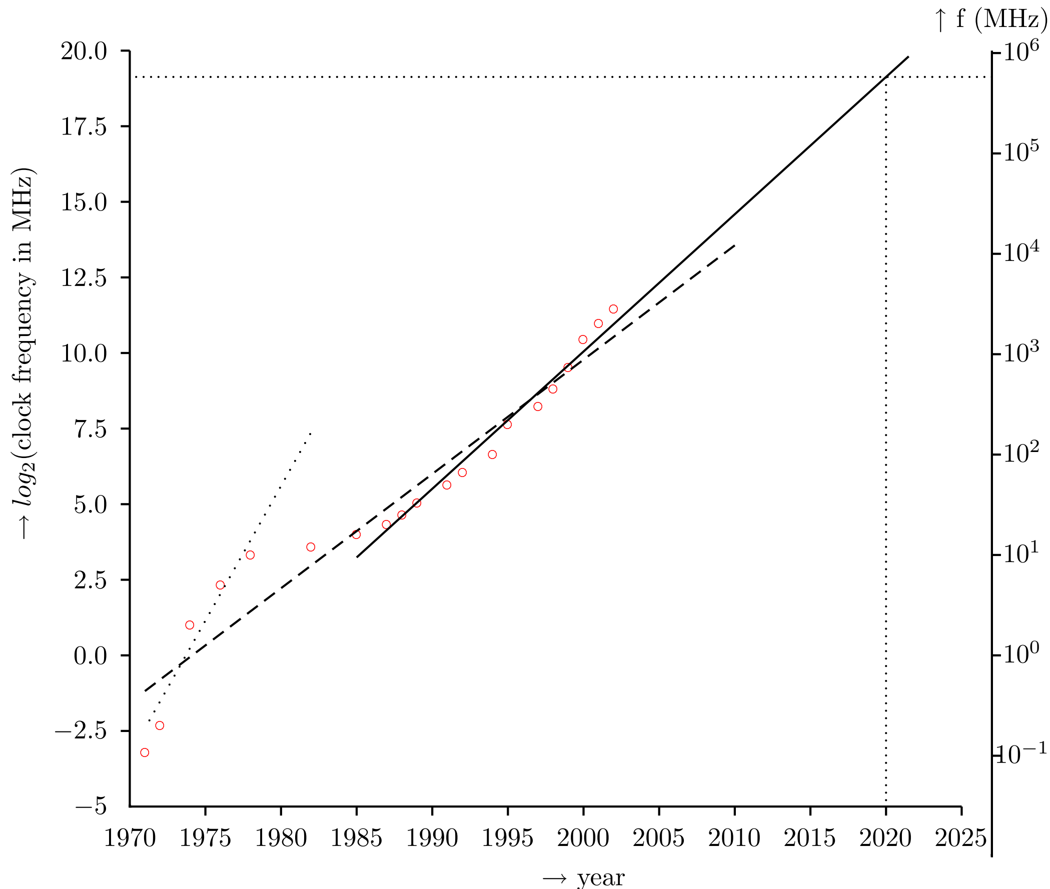
If the amount of energy dissipated per logical operation would have been constant over the last thirty years computers would melt themselves by now. The energy dissipation has fortunately decreased as the number of transistors per chip increased. The energy dissipation can however not decrease forever, thermal noise will put a limit on the lowest possible amount of energy dissipated per logical operation. Thermal noise is of the order of magnitude of 1 kT, where k is the Boltzmann constant, and T absolute temperature. Extrapolation [9] of the current trend of the decrease of energy dissipation per logical operation indicates that the 1 kT level will be reached around the year 2020.

As mentioned before, the number of transistors on a single chip has increased exponentially, while at the same time the physical size of these chips has remained almost constant. As a consequence, the number of atoms used to store one bit of information has decreased exponentially. It seems that this trend has to come to an end at the level of one atom per bit of information, but presumably already before this level is reached. Extrapolation of this trend[9] predicts that the level of one atom per bit of information will be reached in 2020 too.

By extrapolating the exponential increase of the clock frequency to the year 2020 it is possible to obtain an estimate of the frequency at which no more increase is possible. Figure 3.2 contains a plot of the base 2 logarithm of the clock frequency of Intels microprocessors as a function of time. The necessary data has been obtained from Intel, see `http://www.intel.com/pressroom/kits/quickrefyr.htm`.

The increase in clock frequency does not follow an exponential law as accurate as was the case for the increase of the number of transistors per chip in figure 3.1. The dashed line is again a least square fit to a straight line. Especially there is a hop in the data at around 1980, shifting from one exponential trend to another. The dotted line is a fit to the first 5 data points, and the solid line a fit to the last 14 data points. The agreement of the

data points with these two partial fits is good, especially the second one, applying from 1985, and onwards. The point at 1982, which was excluded from the above two partial fits, corresponds to the introduction of the 80286 processor.



o	clock frequency of processor
· · ·	least square fit to straight line first 5 points
- - -	least square fit to straight line all points
—	least square fit to straight line last 14 points

Figure 3.2: *Processor clock frequency*

If the trend from 1985 to 2002 (solid line) is extrapolated to the year 2020, it is found that processors by that time will operate at a clock speed of 570 GHz. Extrapolating exponential trends is however delicate because small changes in the data give enormous changes in the extrapolated values. Other predictions[9] say that in the year 2020 computers will

operate at a clock speed of approximately 40 GHz, have 160 Gbyte of memory, and have a power consumption of 40 W.

It is difficult to say something about the accuracies of the above mentioned figures. Certainly computers will continue to increase their clock speed and amount of memory for a while. But in the not too distant future these trends have to stop on physical grounds.

3.2 More physical limitations

The Turing machine provides for a general and universal model of computation. By means of the Turing machine it is possible to give a precise definition of an algorithm and assess the question of which problems are solvable by means of a computer and which problems are not. The Turing machine is however a mathematical abstraction and does not take physical reality into account.

Computers consist of material components, and the fact that the number of particles in the universe, although large, is finite means that there is a limit to what can be computed. For example, the infinite tape that is used in the Turing machine has no counterpart in the real world because of the finite number of particles in the universe. Even though such considerations have no immediate consequences for practical computing they do have consequences for the Turing machine as a model of computation.

In section 3.1 it has been shown that the current trend of miniaturization of integrated circuits will come to an end within a few decennia, because the way computers are constructed do not allow the individual components to be smaller than approximately the size of a Si-atom. Moreover, at the atomic level physical reality can only be understood in terms of a physical theory called quantum mechanics. The quantum theory, the foundations of were laid in the first half of the twentieth century, is the most accurate model of reality that exists. Hence, any model of computation should ultimately be based upon quantum mechanics. In other words a valid theory of computation must take into account what is

physically realizable, and cannot only be based upon what is mathematically feasible.

It has for example been proposed that the Church Turing thesis should be replaced, such that quantum mechanics is taken into account [25]. This issue will not be considered here, but a number of consequences of quantum mechanics for computing will be considered later.

3.3 Energy dissipation and reversibility

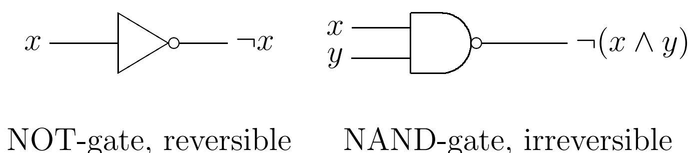
3.3.1 Logical gates

In section 3.1 it was already mentioned that the energy dissipation per logical operation had to decrease in order to sustain further miniaturization of components, otherwise the ICs would simply melt. This has raised the question whether or not energy *has* to be dissipated on logical operations. The answer was given in 1961 by R. Landauer [23], who showed that energy has to be dissipated if and only if information is lost during the operation. Moreover, the amount of dissipated energy is at least kT per bit of lost information.

If information gets lost during a logical operation, then the operation is *irreversible*. The AND operation is an example of an irreversible logical operation because the output 0 corresponds to three possible inputs. Because of the loss of information it is not possible to determine from which of these three inputs the output 0 resulted. The NOT operation, on the other hand, is reversible, because the input can always be determined from the output by negating the output. The NAND-gate, which logically is an AND-gate followed by a NOT-gate also is irreversible.

3.3.2 Universal gates

The NAND-gate is a universal gate, meaning that any function on bits can be computed by a circuit consisting of NAND-gates only. Another way of saying this is that all logical



NOT-gate, reversible NAND-gate, irreversible

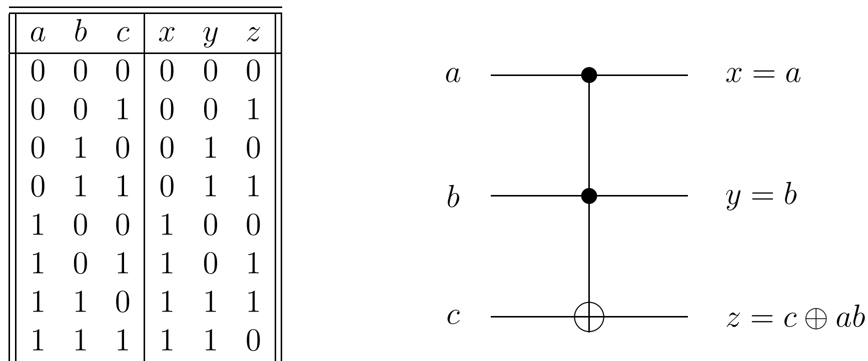
Figure 3.3: *Reversible and irreversible gates*

circuits can be built out of NAND-gates only. The NAND-gate is however not reversible. Three different inputs all yield the output 1, which make it generally impossible to infer the input from the output. As a consequence the NAND-gate, and all logical circuits constructed out of them, must dissipate energy.

At this point the question is: does a logical gate exist that is both universal and reversible. This question is not only interesting in its own right, but also is related to the question what computers can achieve and what they can not achieve, which was taken up in chapter 2. Logical circuits, being an abstraction of real computers, comprise an alternative model of computation, that allow to model and analyze the properties of real computers. As a matter of fact logical circuits, as a model of computation, have been shown to be equivalent to Turing machines, see for example reference [26]. Hence, for any Turing machine there exists a logical circuit that can perform the same computation as the Turing machine.

In 1982 Fredkin and Toffoli showed that the laws of (classical) physics do permit for a logical gate that is both universal and reversible [24]. They also constructed such gates, one of them is the Toffoli gate, see figure 3.4

The input bits are denoted a, b, and c, and the output bits x, y, z. The input bits a, and b are copied to output unchanged, whereas the output bit $z = c \oplus ab$. The product ab is identical to the logical operation a AND b. The symbol $\oplus$ denotes the exclusive or operation. In words, the z-output bit is obtained by taking the exclusive or of c with the result of the and-operation on a and b. The bits a, and b are also called control bits,



a	b	c	x	y	z
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	1
1	1	1	1	1	0

Figure 3.4: *Toffoli gate: reversible and universal.*

because they do affect the result of the entire gate but are unchanged themselves. The main function of theses control bits is that these allow for the reversibility of the logical operation. Table 3.5 illustrates that given ab and $c \oplus ab$ the input value of c can uniquely be inferred. For example, if $ab = 1$, and $c \oplus ab = 0$, then there is only one row in the table matching these values, the last one in this case, giving $c = 1$.

ab	c	$c \oplus ab$	$(c \oplus ab) \oplus ab$
0	0	0	0
0	1	1	1
1	0	1	0
1	1	0	1

Figure 3.5: *Toffoli gate is its own inverse.*

The table also illustrates that the Toffoli gate is its own inverse. The input bits a and b are unchanged anyway. The input bit c results in $c \oplus ab$ after the first Toffoli gate, and in $c = (c \oplus ab) \oplus ab$ after the second Toffoli gate. The second and fourth columns in the table show that $c = (c \oplus ab) \oplus ab$, and hence two Toffoli gates in a row produces output bits identical to the input bits.

3.4 How to compute quantum mechanically

3.4.1 The classical bit

The all important entity in computing is the concept of a *bit*. The word itself came into existence as a short form for the the two word notion of a *binary digit*. Usually with a *bit* is meant the physical realization of a binary digit in hardware. The binary number system is in a way the simplest number system, because it contains only two tokens, the 0, and the 1. Nevertheless, all numbers and strings of characters, and hence all information can be represented in the binary number system. In other words, each piece of information can be translated into a sequence of zeros and ones. Turing machines are usually designed to handle information in the binary number system, although this is not essential.

In real computers binary digits are represented by magnetic poles, or by voltage-levels of electronic components, and termed bits. These bits are the fundamental units to be processed in a logical electronic circuit. As already mentioned are logical circuits equivalent to Turing machines as models of computation.

3.4.2 The quantum bit

Fundamental to quantum computing is the two state quantum system. Examples of two state quantum systems are the polarization direction of a photon and the spin state of an electron. A two state quantum system, as part of a quantum computer, is called a quantum bit, or a qubit.

Quantum systems behave fundamentally different from classical systems. A two state system has only two states that can be measured, but before the measurement is done the system can be in what is called a superposition of these two states. Mathematically this means that the state $|\psi\rangle$ of a two state system is a linear combination of the two states

$|\psi_0\rangle$, and $|\psi_1\rangle$. These latter two states are termed eigenstates. Thus

$$|\psi\rangle = c_0|\psi_0\rangle + c_1|\psi_1\rangle \tag{3.1}$$

The numbers c_0, and c_1 are generally complex numbers. The notation $|\ \rangle$ is called Dirac notation and states such as $|\psi\rangle$ can be thought of as abstract states or as mathematical functions. All these functions should be normalized to 1. As a consequence the numbers c_0, and c_1 must satisfy

$${c_0}^2 + {c_1}^2 = 1. \tag{3.2}$$

Since multiplication of a complex number z with a phase factor $e^{i\phi}$ does not change the modulus of that number, i.e.,

$$|z| = |e^{i\phi}z| \tag{3.3}$$

the state $|\psi\rangle$ from equation 3.1 may be multiplied by such a phase factor without changing its physical meaning. This property can be used to eliminate one more degree of freedom of the parameters c_0, and c_1. For example, one can choose a factor $e^{i\phi}$ such that $e^{i\phi}c_0$ is a real number. This requirement, together with the normalization condition in equation 3.2 eliminates two of the four degrees of freedom of the system of two complex numbers c_0 and c_1, leaving two degrees of freedom.

The system $|\psi\rangle$, representing a qubit can be visualized as a unit vector in a three-dimensional real space, see figure 3.6.

Whereas a classical two-state system really has two states only, a quantum two-state system can be in any out of an infinite number of states, even though only two of them can be measured, namely the states $|\psi_0\rangle$, and $|\psi_1\rangle$. On measurement, the probability of finding the system in state $|\psi_0\rangle$ is $|c_0|^2$, and the probability of finding the system in state $|\psi_1\rangle$ is $|c_1|^2$.

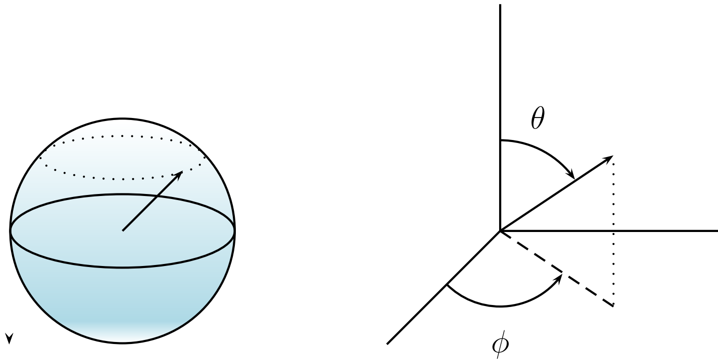


Figure 3.6: *qubit*

The two parameters characterizing the state of the qubit can be chosen as the angles θ and ϕ, shown in figure 3.6. If the states $|\psi_0\rangle$, and $|\psi_1\rangle$ are denoted $|0\rangle$, and $|1\rangle$, for brevity, then the state $|\psi\rangle$ can be written as

$$|\psi\rangle = cos(\frac{\theta}{2})|0\rangle + e^{i\phi}sin(\frac{\theta}{2})|1\rangle \tag{3.4}$$

The state $|0\rangle$ corresponds to $\theta = 0$, in which case the arrow in the sphere points upwards. The state $|1\rangle$ corresponds to the value $\theta = 180°$, corresponding to a downwards pointing arrow. The parameter θ determines the relative proportions of $|0\rangle$ and $|1\rangle$ in the the state $|\psi\rangle$, whereas the parameter ϕ determines their relative phases. For a fixed value of θ different phases correspond to different arrows, ending on the same dotted circle in figure 3.6. This amounts to rotation about the z-axis.

3.4.3 Many qubits

A system consisting of two qubits has $2 \times 2 = 4$ measurable states. These are the states $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$. Here $|00\rangle$ is the same as $|0\rangle|0\rangle$, etcetera. These states, now form

the basis states for forming the total state $|\psi\rangle$ of the system reading

$$|\psi\rangle = c_{00}|00\rangle + c_{01}|01\rangle + c_{10}|10\rangle + c_{11}|11\rangle. \tag{3.5}$$

It is seen that the state of a system of two qubits is specified by the *four* complex numbers c_{00}, c_{10}, c_{01}, and c_{11}. The state $|\psi\rangle$ is a superposition of the states $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$. For a system consisting of n qubits the state $|\psi\rangle$ will be a superposition of the states $|q_1 q_2 \cdots q_n\rangle$ which are 2^n in number because each of the qubits $q_i, i = 1, \cdots n$ can assume the values 0 and 1.

Hence, 2^n numbers are needed to specify the state of the system. In other words, a system of n qubits stores an amount of information of the size of 2^n complex numbers. This amount is an exponential function of the number of qubits, in contrast to the non-quantum case, where the amount of information is linear in the number of bits. For $n = 500$ the number 2^n is larger than the number of particles in the universe.

In summary, a quantum memory is capable of storing a exponential amount of information in a polynomial number of qubits. So far, little is known of how to make use of this potentially vast computational power. It is the challenge of quantum computing to find ways of exploiting this potential.

3.5 Quantum computation

3.5.1 Single qubit gates

In a classical logic circuit the only non-trivial gate that acts on a single bit is the NOT-gate. The NOT-gate simply flips a bit. The quantum analogue to this is to flip a qubit. For a state in an arbitrary superposition this means that

$$
\begin{aligned}
\mathrm{NOT}|\psi\rangle &= \mathrm{NOT}(c_0|0\rangle + c_1|1\rangle) \\
&= c_0\mathrm{NOT}|0\rangle + c_1\mathrm{NOT}|1\rangle = \\
&= c_0|1\rangle + c_1|0\rangle
\end{aligned}
\tag{3.6}
$$

The quantum NOT interchanges the states $|0\rangle$ and $|1\rangle$, but the same result is obtained by interchanging the coefficients c_0, and c_1, as can be seen from equation 3.6.

If theses two coefficients are collected in a column vector the interchange can be accomplished by a matrix-vector multiplication, as follows. Denote the vector representing the state $|\psi\rangle$ as

$$
\mathbf{c} = \begin{pmatrix} c_0 \\ c_1 \end{pmatrix}. \tag{3.7}
$$

Then

$$
\mathrm{NOT}\ \mathbf{c} = \begin{pmatrix} c_1 \\ c_0 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} c_0 \\ c_1 \end{pmatrix} \tag{3.8}
$$

Is is clear that there are many more possible matrix operations acting on a single qubit. Which transformations are allowed and which are not? The answer is that unitary transformations are allowed. Unitary transformations are such that they do not affect the normalization of the coefficients: $c_0^2 + c_1^2 = 1$ both before and after the transformation.

There are infinitely many unitary 2 by 2 matrices, and hence infinitely many logical single qubit gates. Another important one, apart from the NOT gate, is the Hadamard gate, characterized by the matrix

$$
H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{3.9}
$$

The Hadamard gate is such that if acting on a qubit in the state $|0\rangle$ the qubit will afterwards be in a state with equal contributions from the states $|0\rangle$, and $|1\rangle$. Hence,

$$H\begin{pmatrix}1\\0\end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix} \tag{3.10}$$

The Hadamard gate is important because it allows to prepare a qubit, originally in the eigenstate $|0\rangle$, to end up in a superposition with equal contributions from both eigenstates. Such a preparation plays an important role in quantum algorithms, in the form of so called quantum parallelism. The logic circuit symbols for the quantum NOT and Hadamard gates are given in figure 3.7.

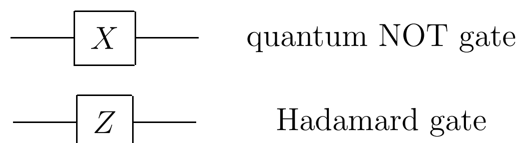


Figure 3.7: *NOT and Hadamard quantum gates*

3.5.2 Multiple qubit gates

It has been pointed out in section 3.3 that the classical AND, OR, and NAND gates are not reversible due to loss of information: the gate throws away information. For a quantum gate to be physically realizable is is absolutely necessary that the gate does not violate the principles of quantum mechanics. In quantum mechanics all evolution, that is, all changes with time, are accomplished through unitary transformations, which are reversible. Therefore only reversible quantum gates are candidates for being physically realized some time in the future. Only reversible quantum gates will therefore be considered.

The most important two-qubit quantum logic gate is the controlled NOT gate, abbreviated CNOT gate[1], shown in figure 3.8. The control qubit, marked a, is always unchanged

[1] By the time of writing of this paper an experimental realization of the quantum CNOT-gate is reported in the october issue of *Nature*, reference [30]

but affects the action on the target qubit, here marked b. If the control qubit equals 0 the target qubit also is unchanged. If the control qubit is 1 the target qubit is inverted. By explicitly writing out one can easily check that the effect on b can be written as the a XOR b, which equals addition modulo 2, $a \oplus b$.

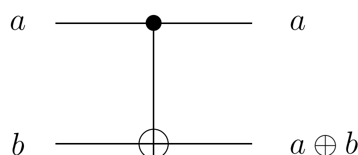


Figure 3.8: *The CNOT quantum logic gate.*

If the first qubit signifies the control qubit, and the second qubit the target qubit, then $|00\rangle$, and $|01\rangle$, will be unchanged by the action of the CNOT gate, whereas $|10\rangle$, and $|11\rangle$, will be interchanged. A system of two qubits generally has a state given in equation 3.5. Acting on that state will interchange $|10\rangle$ and $|11\rangle$, which is the same as interchanging the coefficients c_{10} and c_{11}, and leaving the first two coefficients unchanged. The action of the CNOT gate can therefore equally well be described by a matrix acting on the vector of coefficient, as follows

$$\text{CNOT}\,\mathbf{c} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} c_{00} \\ c_{01} \\ c_{10} \\ c_{11} \end{pmatrix} = \begin{pmatrix} c_{00} \\ c_{01} \\ c_{11} \\ c_{10} \end{pmatrix} \tag{3.11}$$

Single qubit gates, together with the two qubit CNOT gate form a universal set. That is, each valid, i.e., unitary operation on n qubits can be achieved by a quantum logic circuit consisting of single qubit gates and CNOT gates only. Gates involving more than two qubits can be constructed from single qubit gates and CNOT gates. Mathematically

this can be proven by proving that the $2^n \times 2^n$ unitary matrix describing the action on the coefficient vector can be decomposed into the 2×2 unitary matrices acting on a single qubit, and 4×4 CNOT matrices acting on two qubits.

3.5.3 Function evaluation and quantum parallelism

A function on n qubits can be evaluated with a quantum circuit consisting of CNOT and single qubit gates. Making use of superposition it is actually possible to simultaneously evaluate the function for different values of the argument. Consider a function of one bit, $f(x) : \{0,1\} \rightarrow \{0,1\}$. In order to be able to reversibly evaluate the function the following scheme usually is adopted.

A system containing two qubits is needed, because the value of the argument has to be carried through for the system to be reversible. The function evaluation is accomplished by an appropriate circuit of quantum logic gates, and can be done such that the system state $|x, y\rangle$ is transformed into $|x, y \oplus f(x)\rangle$. The y qubit has to be in the output as well, in order to be able to infer the input from the output. If $y = 0$ on input one obtains $|x, 0\rangle \rightarrow |x, f(x)\rangle$. The unitary evaluation corresponding to the evaluation of f is denoted U_f, and the corresponding circuit symbol is given in figure 3.9.

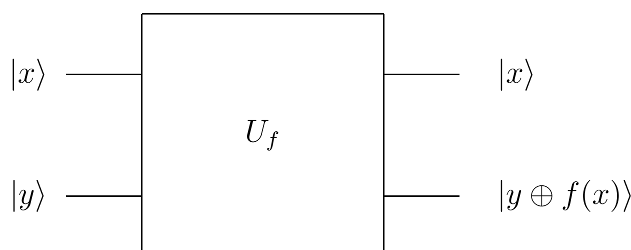


Figure 3.9: *Function evaluation of a function on* 1 *bit.*

If $f(x)$ is the identity function, i.e., $f(0) = 0$, and $f(1) = 1$, the function evaluation can be accomplished by a CNOT gate. This can be verified by explicitly by writing out a table, or by comparing figure 3.9 for function evaluation with figure 3.8 for the CNOT

gate.

The qubit containing the argument x can be prepared in the state $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, which can be done by the acton of the Hadamard gate on $|0\rangle$. If the target bit $y = 0$ then the application of U_f results in the state $\frac{1}{\sqrt{2}}(|0, f(0)\rangle + |1, f(1)\rangle)$ for the entire system consisting of two qubits. The operation is depicted in figure 3.10.

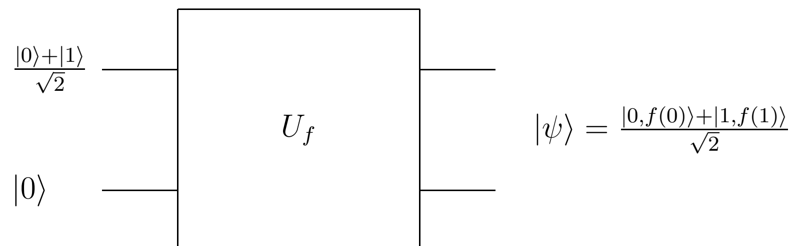


Figure 3.10: *Quantum parallelism.*

It can be seen that the resultant state contains the function values for *both* values of the argument. The function has, however, been evaluated only once. Thus, both argument values have been evaluated *in parallel*. This phenomenon is called quantum parallelism.

The resultant state is a linear combination, which is such that no single qubit can be factored out, such a state is called an entangled state. The linearity of quantum mechanics plays a crucial role in obtaining the resultant entangled state which can be seen as follows. The operator U_f evaluating the function is defined as

$$U_f|x, y\rangle = |x, y \oplus f(x). \tag{3.12}$$

The operation of U_f on the state $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ can be written as

$$
\begin{aligned}
U_f \frac{1}{\sqrt{2}} \sum_{x=0}^{1} |x, 0\rangle &= \frac{1}{\sqrt{2}} \sum_{x=0}^{1} U_f |x, 0\rangle \\
&= \frac{1}{\sqrt{2}} \sum_{x=0}^{1} |x, 0 \oplus f(x)\rangle \\
&= \frac{1}{\sqrt{2}} \sum_{x=0}^{1} |x, f(x)\rangle \\
&= \frac{|0, f(0)\rangle + |1, f(1)\rangle}{\sqrt{2}}
\end{aligned}
\tag{3.13}
$$

The linearity of quantum mechanics allowed to interchange the action of U_f and the summation in the first step. In the next step the definition of U_f was applied after which the result could be rewritten in the same form as in figure 3.10.

Although the function f has been evaluated for both values of the argument in parallel, not both function values can be simultaneously obtained. The reason is that upon reading off the quantum register it will assume one of the two eigenstates that the entangled state is made up of. The observer has no way of affecting this process. Hence, quantum parallelism will have to be complemented in some way in order to become useful. Only a function of one qubit was discussed here, but quantum parallelism also shows up for functions of n qubits. However, also for n qubits, although all n argument values can be evaluated in parallel, only one of them can subsequently be obtained through reading off the qubits. This is a direct consequence of the laws of quantum mechanics that make it impossible to extract more than state from a superposition of states. After the observation of one state, all information about the other states does not exist anymore.

3.5.4 Deutsch's algorithm

At this point the obvious question is how to make use of quantum parallelism, if at all possible. As explained above it is not possible to gain access to more than one of the simultaneously evaluated function values. Instead of function values there might be other, perhaps more global, information about the function, that a quantum computer is able to compute more efficient than a classical computer.

In 1984 David Deutsch [25] demonstrated such a property, see also [27]. Of the functions of one bit $f(x) : 0,1 \to 0,1$, there are four different ones. These can be classified as constant, or balanced. The two constant functions are $f(x) = 0$, and $f(x) = 1$. The remaining two are balanced in the sense that the values 0, and 1 appear equally often as function values. These functions are $f(0) = 0, f(0) = 1$, and $f(0) = 1, f(0) = 0$.

In order to determine whether or not a one bit function is constant or not two function evaluations are necessary. The property itself is, however, a one-bit property in the sense that there are only two possible values: constant or balanced. A quantum computer can evaluate the function in parallel for the two different argument values. The problem is to further process the information in such a way that once the value of the qubit is read off this will give relevant information about the function being constant or balanced.

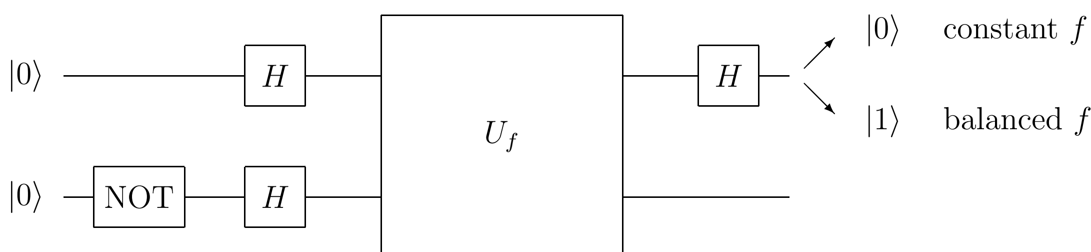


Figure 3.11: *Circuit determining balanced or constant.*

A quantum logic circuit solving the problem is given in figure 3.11. The input is a two qubit system $|x, y\rangle$, where both x, and y are 0. First y is set to 1 through an inverter, after

which a Hadamard transformation is applied to both x, and y.

The Hadamard transformation turns $|0\rangle$ into $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle$, and $|1\rangle$ into $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle$. Thus before being processed by U_f the state of the two qubits is $\frac{1}{2}\sum_{x=0}^{1}|x\rangle(|0\rangle - |1\rangle)$. It can be verified that the application of U_f results in the state $\frac{1}{2}\sum_{x=0}^{1}(-1)^{f(x)}|x\rangle\,(|0\rangle - |1\rangle)$, see appendix A.1 for details. Thus the application of U_f has left the qubit y untouched. The interesting part is the x qubit, the state of which depends on the function f. If f is constant x contains $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$, and if f is balanced x contains $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. If now again a Hadamard transformation is applied on x the content of x is $|0\rangle$ for a constant function f, and $|1\rangle$ for a balanced function f.

The procedure can readily be generalized to functions of n bits, with only the values 0 and 1 as the allowed function values. In that case a function is not necessarily either constant or balanced, there exist also other possibilities, these are not considered. It can be shown that for these functions of n bits the property balanced or constant can be determined with one parallel quantum evaluation of the function, thereby achieving an exponential speedup as compared to determining this property classically, which requires more than 2^{n-1} function evaluations, and which obviously is exponential in the size of the input.

So far only few problems have been demonstrated to be able to profit, in the sense of speed up of execution, from quantum parallelism. The above described problem by Deutsch was the first one. Although the practical usefulness is questionable, the problem does show that there exist problems that would gain from being executed on a quantum computer. We will discuss three more interesting problems that would profit from being executed on a quantum computer, these are the Fourier transform [21], the integer factoring problem [21], and the search in a database [22]. The Fourier transform plays a rather central role in the relatively few existing quantum algorithms that solve problems with a time complexity that compares favourable to algorithms that solve the same problem on a conventional computer.

3.6 The Fourier transform

3.6.1 Mathematical Definition

The Fourier transform is a mathematical operation that transforms a function into another function, and can be done for most functions that behave 'reasonable'. Here, the Fourier transform of functions of one variable will be considered. The Fourier transform has applications in many different fields, often using their own specific notation. If the function to be transformed is a function of time, denoted t, the transformed function is a function of frequency, denoted f. The transformed function, denoted $H(f)$, can be considered as a decomposition of the original function, denoted $h(t)$, into frequency components, where the transformed function gives the amplitude of each frequency component. The relation between $H(f)$ and $h(t)$ is

$$H(f) = \int_{-\infty}^{\infty} h(t) e^{2\pi i f t} dt \tag{3.14}$$

3.6.2 The Discrete Fourier transform

The discrete Fourier transform is obtained if the continuous integral in equation 3.14 is replaced by a discrete sum, and can be viewed as an approximation to the Fourier transform. The discrete Fourier transform is especially useful if the function to be transformed consists of a set of discrete data $h_k, k = 1 \cdots N$, obtained by, for example, some measurement. The transformed function consists in that case of a discrete set of data as well, denoted as $H_n, n = 1, \cdots N$, containing as many, N, elements, as the original set. The two sets are related by

$$H_n = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} h_k e^{k \frac{2\pi i}{N} n}. \tag{3.15}$$

Note that in equation 3.15 the data is numbered from $0 \cdots N - 1$, instead of $1 \cdots N$.

For a discrete set of data the discrete Fourier transform can just as well be considered as a transformation in its own right, without being viewed as an approximation of the continuous Fourier transform.

Often the data to be transformed is a large set of sampled data, therefore it is of interest to consider the time complexity of the discrete Fourier transform. From equation 3.15 it can be deduced that N numbers H_n have to be calculated, and for each number one has to sum N terms where for each term a multiplication has to be performed. This is seen most easily by defining

$$W = e^{\frac{2\pi i}{N}} \tag{3.16}$$

and rewriting 3.15 as

$$H_n = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} W^{nk} h_k. \tag{3.17}$$

The complexity of the discrete Fourier transform, if performed according to formula 3.17, i.e, a matrix times vector multiplication, is therefore $O(N^2)$.

For a general matrix times vector operation the time complexity cannot be reduced below $O(N^2)$. The form of the matrix used in the discrete Fourier transform, W^{nk}, is such that it actually is possible to do much better than $O(N^2)$.

3.6.3 The Fast Fourier transform

The fast Fourier transform is an algorithm for performing the discrete Fourier transform, which utilizes the special structure of the transformation coefficients W^{nk}, see formula 3.17. This results in an algorithm with time complexity $O(N\log_2(N))$ instead of $O(N^2)$.

Not seldom, if a complexity factor N is replaced by a factor $\log_2(N)$, this is the result of rephrasing the problem from an iterative into a recursive fashion. Examples are the binary search algorithm, and the quicksort algorithm. Also the fast Fourier transform can

be phrased in a recursive manner, as follows. The set of N points h_k can be partitioned into two subsets of size $N/2$, one subset containing the even numbered data points, and the other one containing the odd numbered points. Assuming for simplicity that N is a power of 2, the discrete Fourier transform can be written as,

$$H_n = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{n \frac{2\pi i}{N} k} h_k \tag{3.18}$$

$$= \frac{1}{\sqrt{N}} \sum_{k=0}^{N/2-1} e^{n \frac{2\pi i}{N} 2k} h_{2k} + \frac{1}{\sqrt{N}} \sum_{k=0}^{N/2-1} e^{n \frac{2\pi i}{N} (2k+1)} h_{2k+1} \tag{3.19}$$

$$= \frac{1}{\sqrt{N}} \sum_{k=0}^{N/2-1} e^{n (\frac{2\pi i}{N/2}) k} h_{2k} + W^n \frac{1}{\sqrt{N}} \sum_{k=0}^{N/2-1} e^{n (\frac{2\pi i}{N/2}) k} h_{2k+1} \tag{3.20}$$

$$= \frac{1}{\sqrt{N}} \sum_{k=0}^{N/2-1} (W^2)^{nk} h_{2k} + W^n \frac{1}{\sqrt{N}} \sum_{k=0}^{N/2-1} (W^2)^{nk} h_{2k+1} \tag{3.21}$$

In equation 3.19 the summation has been split into a summation over even, $2k$, and odd, $2k+1$ values for the summation index. In 3.20 a factor W^n has been taken out of the second summation, and the extra factor 2 results in N being replaced by $N/2$ in the exponential factors. Hence, both summations are nothing else than a discrete Fourier transform of length $N/2$, which also can be written in the form of 3.21. Note that W has periodicity N, meaning that $W^N = 1$, and W^2 has periodicity $N/2$, as required.

Having reduced the discrete Fourier transform to two discrete Fourier transforms of half the length, the same procedure can be applied to both these transforms of half the length. After $\log_2(N)$ divisions one arrives at a discrete Fourier transform of length 1, which is the identity operation.

From equation 3.21 is is deduced that obtaining the discrete Fourier transform from the two transforms of half the length requires 1 addition, and 1 multiplication for each H_n, apart form the calculation of the factor W^k, which has to be done once. This procedure has to be repeated $\log_2(N)$ times, resulting in an overall complexity of $O(N\log_2(N))$.

There is a hook, though, for in order to make the procedure practical the transform is carried out in the opposite direction: first performing transforms of length 2, followed by transforms of length 4, and so on, up to transforms of length N. The h_k that are combined at each level of transformation are spread out through the whole array. It is of course possible to look up the appropriate h_k at each transformation. It is also possible to reorder the h_k once and for all by for each h_k writing the bits of k in reversed order, and putting the original value of h_k at the position of the corresponding number. For example, for $N = 16$, the number $h_{13} = h_{1101}$ will be put at position $h_{1011} = h_{11}$. Fortunately, the procedure of reordering the h_k in bit reversed order has time complexity no greater than $O(N\log_2(N)$, making the overall complexity $O(N\log_2(N)$ as well.

The assumption made above that N should be a power of 2 is no limitation, because a discrete Fourier transform of arbitrary length can be padded with zeros up to the nearest, larger, power of 2. This affects the computational time with a multiplicative factor less than 2 and does therefore not affect the computational complexity.

3.6.4 The Quantum Fourier transform

The discrete Fourier transform replaces each number out of a set by a linear combination of all the numbers in the set. The transformation as defined in equation 3.15 is in fact unitary. The inverse is given by the same transformation, except that the complex factor i is replaced by $-i$. Because the transformation is obviously symmetric this means that the inverse transformation matrix is equal to the complex conjugate of the transposed transformation, which is the definition of a unitary matrix.

In order to explicitly show the unitarity of the discrete Fourier transform we calculate the product of the transformation matrix with its inverse and show that the result equals unity.

$$\frac{1}{\sqrt{N}}\sum_{k=0}^{N-1} e^{j\frac{2\pi i}{N}k}\frac{1}{\sqrt{N}}e^{k\frac{-2\pi i}{N}l} = \frac{1}{N}\sum_{k=0}^{N-1} e^{k\frac{2\pi i}{N}(j-l)} \tag{3.22}$$

$$= \begin{cases} \frac{1}{N}\sum_{k=0}^{N-1} e^0 = 1, \text{ if } j = l \\ \frac{1-e^{\frac{2\pi i}{N}(j-l)N}}{1-e^{\frac{2\pi i}{N}(j-l)}} = 0, \text{ if } j \neq l \end{cases} \tag{3.23}$$

$$= \delta_{jl} \tag{3.24}$$

In equation 3.23 the closed form expression of the summation of a geometric series has been used, see also appendix A.2. The unitarity of the discrete Fourier transform is important because that is a precondition for being able to implement this transformation as a quantum logic circuit, because such a circuit always has to implement unitary transformations.

The discrete Fourier transform as an approximation of the full, continuous transform, is a one-dimensional transform since it transforms a function of one variable into another function of one variable. Fourier transforms of functions of more than one variable can also be defined, these are sometimes called many-dimensional Fourier transforms.

However, as is clear from equation 3.22, already the one-dimensional discrete Fourier transform is a unitary transformation in an N-dimensional space. A transformation of a vector is equivalent to the transposed transformation on the basis vectors. In the case of the discrete Fourier transform, which is symmetric, this means that a transformation on a vector is equivalent to the same transformation on the basis.

Mathematically this can be expressed as follows. If the basis is denoted $|0\rangle, \ldots, |N-1\rangle$ then the transformation of the basis is given by

$$\mathrm{DFT}_N : |j\rangle = \frac{1}{\sqrt{N}}\sum_{k=0}^{N-1} e^{j\frac{2\pi i}{N}k}|k\rangle \tag{3.25}$$

where $\mathrm{DFT}_N : |j\rangle$ stands for the transformed basis vector $|j\rangle$ as a result of the operator

DFT_N.

If now DFT_N : acts on $\sum_{j=0}^{N-1} f(j)|j\rangle$, which is a vector in the space spanned by the given basis, one obtains

$$\mathrm{DFT}_N : \sum_{j=0}^{N-1} f(j)|j\rangle = \sum_{j=0}^{N-1} f(j) \ \mathrm{DFT}_N : |j\rangle \tag{3.26}$$

$$= \sum_{j=0}^{N-1} f(j) \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{j\frac{2\pi i}{N}k}|k\rangle \tag{3.27}$$

$$= \sum_{k=0}^{N-1} \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} e^{k\frac{2\pi i}{N}j} f(j)|k\rangle \tag{3.28}$$

$$= \sum_{k=0}^{N-1} \tilde{f}(k)|k\rangle \tag{3.29}$$

where $\frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} e^{k\frac{2\pi i}{N}j} f(j) = \tilde{f}(k)$ is the discrete Fourier transform of the function $f(j)$ in the interval $0 \ldots N-1$. The $f(k)$ are what was called h_k, and the $\tilde{f}(k)$ what was called H_k.

Equation 3.29 has the same form as the state of a quantum register as described in section 3.4.3 for two qubits. Hence, equation 3.29 also shows that by performing a discrete Fourier transform on the qubits of a quantum register one effectively performs a discrete on the whole vector $f(k)$, in one sweep. It has already been shown that the discrete Fourier transform is unitary. Therefore, a quantum circuit that performs such a transform on the qubits can be constructed. For the quantum Fourier transform to be useful, this circuit must consist of a number of logic gates, that is polynomial in N. It will be shown that this condition can be fulfilled by explicitly constructing such a circuit.

The starting point for the construction is equation 3.25 which describes how a basis state transforms. The index k, numbering the basis states, is made up of n qubits, hence k ranges from $0 \ldots N-1 = 0 \ldots 2^n - 1$. The bits of the number k are ordered with the

least significant bit on the right, and the least significant bit on the left. This still leaves many possible ways of numbering the bits, one possibility is

$$k = (k_{n-1}k_{n-2}\dots k_1k_0)_2 = k_0 2^0 + k_1 2^1 + \dots + k_{n-1}2^{n-1} = \sum_{l=0}^{n-1} k_l 2^l \tag{3.30}$$

where the notation $k = (k_{n-1}k_{n-2}\dots k_1k_0)_r$ stands for the number k written in the the positional number system with base, or radix, r. It is the position of the digit that determines to which power of r it is the coefficient, where the rightmost digit always is the coefficient of r^0. By numbering the digits as in equation 3.30 the power of r is identical to the index of the digit.

The suffix r indicating the radix will be left out because the base-2 number system will be used throughout. Here it is more convenient to use a different numbering of the digits, namely

$$k = (k_1k_2\dots k_n) = k_1 2^{n-1} + k_2 2^{n-2} + \dots + k_n 2^0 = \sum_{l=1}^{n} k_l 2^{n-l} \tag{3.31}$$

Note that still the position decides the significance of a bit: k_n, appearing as the rightmost digit is the least significant.

A quantum register containing the number k can be written in each of four equivalent ways

$$|k\rangle = |k_1k_2\dots k_n\rangle = |k_1\rangle|k_2\rangle\cdots|k_n\rangle = \bigotimes_{l=1}^{n} |k_l\rangle \tag{3.32}$$

These notational issues being established the discrete Fourier transform can be written as, writing k in binary form and using that $N = 2^n$,

$$
\begin{aligned}
\mathrm{DFT}_N : |j\rangle =& \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{j\frac{2\pi i}{N}k} |k\rangle & (3.33)\\
=& \frac{1}{\sqrt{N}} \sum_{k_1=0}^{1} \sum_{k_2=0}^{1} \cdots \sum_{k_n=0}^{1} e^{j\frac{2\pi i}{N}\sum_{l=1}^{n} k_l 2^{n-l}} |k_1 k_2 \ldots k_n\rangle & (3.34)\\
=& \frac{1}{\sqrt{N}} \sum_{k_1=0}^{1} \sum_{k_2=0}^{1} \cdots \sum_{k_n=0}^{1} \bigotimes_{l=1}^{n} e^{j2\pi i k_l 2^{-l}} |k_l\rangle & (3.35)\\
=& \frac{1}{\sqrt{N}} \bigotimes_{l=1}^{n} \left[\sum_{k_1=0}^{1} e^{j2\pi i k_l 2^{-l}} |k_l\rangle \right] & (3.36)\\
=& \frac{1}{\sqrt{N}} \bigotimes_{l=1}^{n} \left[|0\rangle + e^{j2\pi i 2^{-l}} |1\rangle \right] & (3.37)
\end{aligned}
$$

First the summation over k in equation 3.33 has been rewritten as a repetitive summation over the individual bits of k, using the representation of equation 3.31 in the exponent, after which equation 3.34 has been obtained. Note that the summation over l in the exponent is a formal expansion, whereas the outer summations sum over *values*, 0 and 1, of the individual k_l. Now the divisor N cancels the factor 2^n, and the exponent of a sum of terms can be rewritten as a product of individual exponents, at the same time using 3.32, which gives equation 3.35. Exchanging sums and the direct product gives equation 3.36, and explicitly performing the summation over k_l gives equation 3.37. In doing this last summation, it is only $|1\rangle$ that obtains a pre-factor different from 1. It is actually this fact that is responsible for the $N\log_2(N)$ complexity of the fast Fourier transform, because in each of the $n = \log_2(N)$ factors in equation 3.37 there now is 1 multiplication. If the factor of $|0\rangle$ was different from 1, there would be $2^{\log_2(N)} = N$ multiplications for each j, producing N^2 overall complexity.

The product in equation 3.37 can be written out explicitly, where the fraction $j/2^l$ is

rewritten as

$$\frac{j}{2^l} = \frac{j_1 2^{n-1} + j_2 2^{n-2} \dots j_n 2^0}{2^l}. \tag{3.38}$$

For each value of l only those bits of j where the final exponent of 2 is smaller than 0, i.e., negative, are relevant, because $e^{2\pi i 2^p} = 1$ for $p \geq 0$. Thus, for the $n = 1$ factor, only the j_n bit contributes, for $n = 2$, the j_n, and j_{n-1} bit contribute, and so on. Hence,

$$\text{DFT}_N : |j\rangle = \frac{|0\rangle + e^{2\pi i \frac{j_n}{2}}|1\rangle}{\sqrt{2}} \frac{|0\rangle + e^{2\pi i (\frac{j_{n-1}}{2} + \frac{j_n}{4})}|1\rangle}{\sqrt{2}} \dots \frac{|0\rangle + e^{2\pi i (\frac{j_1}{2} + \frac{j_2}{4} + \dots \frac{j_n}{2^n})}|1\rangle}{\sqrt{2}} \tag{3.39}$$

Expression 3.39 is the form which lends itself for translation into a quantum circuit. For the first factor, it holds that

$$\frac{|0\rangle + e^{2\pi i \frac{j_n}{2}}|1\rangle}{\sqrt{2}} = \begin{cases} \frac{|0\rangle + |1\rangle}{\sqrt{2}} & \text{, if } j_n = 0 \\ \frac{|0\rangle - |1\rangle}{\sqrt{2}} & \text{, if } j_n = 1 \end{cases} \tag{3.40}$$

which is exactly what a Hadamard transformation does on a qubit $|j_n\rangle$ for the two possible cases $|j_n = 0\rangle$, and $|j_n = 1\rangle$.

The value of the second factor depends on the values of the bits j_{n-1} and j_n, and is given in table 3.2. If $j_n = 0$, the effect is a Hadamard transformation on the bit j_{n-1}, just as in the case of the first factor, but now on the bit j_{n-1} instead of j_n. The effect of the bit j_n now is an extra factor i, but, firstly only if $j_n = 1$, and secondly, only on the state $|1\rangle$, as can be seen from the table.

All of the following factors in equation 3.39 are now obtained in a similar way. The p-th factor follows from a Hadamard transformation on qubit $n - p + 1$ plus extra factors $e^{2\pi i \frac{j_{n-p+q}}{2^q}}$ for $q = 2 \dots p$ on the $|1\rangle$ component of the result. Each extra factor only applies if the corresponding bit $j_{n-p+q} = 1$. In a logic circuit the conditional multiplication with these factors can be achieved through controlled single qubit gates, as follows.

j_{n-1}	j_n	$\frac{\lvert 0\rangle + e^{2\pi i(\frac{j_{n-1}}{2}+\frac{j_n}{4})}\lvert 1\rangle}{\sqrt{2}}$
0	0	$\frac{\lvert 0\rangle+\lvert 1\rangle}{\sqrt{2}}$
0	1	$\frac{\lvert 0\rangle+i\lvert 1\rangle}{\sqrt{2}}$
1	0	$\frac{\lvert 0\rangle-\lvert 1\rangle}{\sqrt{2}}$
1	1	$\frac{\lvert 0\rangle-i\lvert 1\rangle}{\sqrt{2}}$

Table 3.2: *Factor in quantum Fourier transform*

The unitary transformation

$$R_q = \begin{pmatrix} 1 & 0 \\ 0 & e^{\frac{2\pi i}{2^q}} \end{pmatrix} \tag{3.41}$$

is a single qubit transformation that multiplies the $|1\rangle$ component of a single qubit state with the factor $e^{\frac{2\pi i}{2^q}}$. In analogy with the controlled NOT gate the controlled R_q gate R_{pq}, acting on two qubits, can be defined as

$$R_{pq} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{\frac{2\pi i}{2^q}} \end{pmatrix} \tag{3.42}$$

which applies the R_q operation if $j_p = 1$. The symbol for the controlled R_q gate is given in figure 3.12.

The controlled R_{pq}-gate exactly performs the conditional multiplication as required in

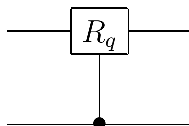


Figure 3.12: *Controlled R_p gate*

the discrete Fourier transform.

A circuit performing the quantum Fourier transform on a set of n qubits is given in figure 3.13. This circuit is a quantum logic implementation of equation 3.39, but not a complete implementation, because the result is in bit reversed order. Hence, elements for performing the qubit-reversion have to be added, these have been left out for conciseness. A circuit for swapping two qubits will be given below.

The input qubits $|j_1\rangle \cdots j_n\rangle$ in figure 3.13 are given in the order according to equation 3.31, with the least significant qubit j_n on the right. The Hadamard transformation on $|j_n\rangle$ produces the first, i.e., leftmost, factor on the right hand side of equation 3.39, and shows up on the rightmost side below in figure 3.13.

The second factor in equation 3.39 involves the qubits j_{n-1}, and j_n, and shows up as the one but rightmost factor on the output side in figure 3.13, and is the result of a Hadamard transformation and a controlled R_2 operation, as explained above.

The input qubit $|j_p\rangle, p+1 \dots n-1$ is acted on by a Hadamard transformation, followed by $n-p$ controlled R_q operations for $q = 2 \dots n-p+1$, where operation R_q is controlled by j_{p+q-1}. Note that, because the set of transformations done on $|j_1\rangle$ is dependent on qubits $|j_2\rangle \cdots j_n$, first all operations on $|j_1\rangle$ are done, then all operations on $|j_2\rangle$, and so on, the Hadamard transformation on $|j_n\rangle$ being done last.

In order to reverse the qubits that result from the circuit in figure 3.13 some more circuitry is needed. Figure 3.14 shows how two qubits can be swapped, by the repeated application of the CNOT-gate. Recall that $a \oplus b$ is binary addition modulo 2, which is equivalent to the XOR operation, hence, $(a \oplus b) \oplus a = b$.

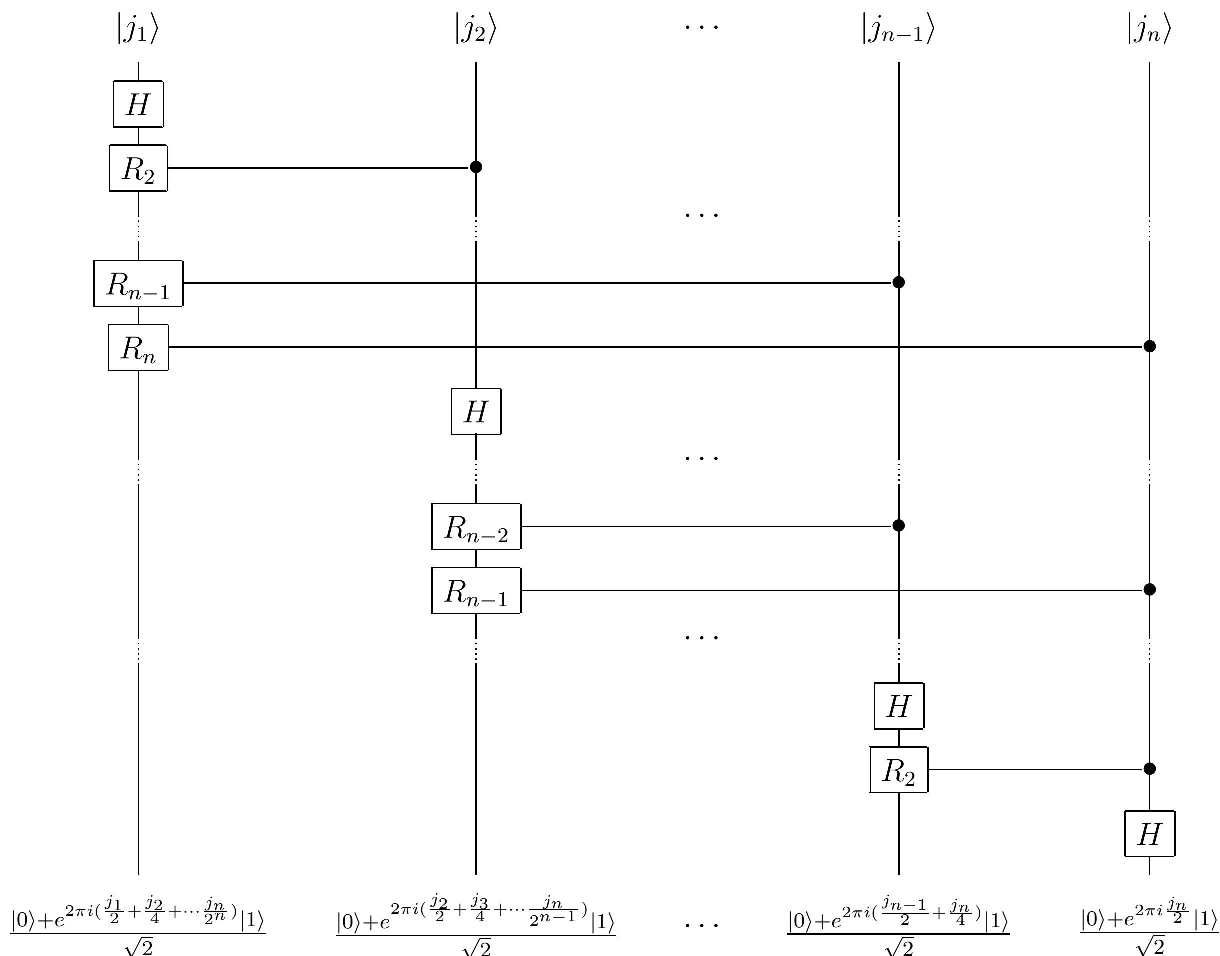


Figure 3.13: *Circuit performing the quantum Fourier transform*

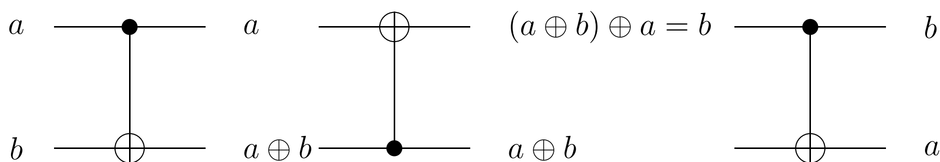


Figure 3.14: *Swapping two qubits.*

If the three operations shown in figure 3.14 are done immediately after each other the circuit shown in figure 3.15 is obtained.

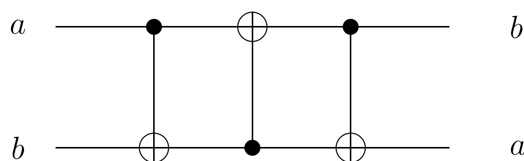


Figure 3.15: *Logic circuit swapping two qubits.*

The number of components needed in a circuit to perform the quantum Fourier transform is polynomial in the number of qubits, n. This can easily be established by counting the number of gates.

There are n Hadamard gates. Then there are $n-1+n-2+\ldots+1=n(n-1)/2$ conditional R_q gates. Furthermore there are $n/2$ swap-gates of the type shown in figure 3.15 needed, in order to perform the final bit reversion. Since all of these are polynomials in n the size of the final circuit also is polynomial in n.

As a consequence the quantum Fourier transform has a computational complexity of $\text{poly}(\log_2(N))$, in contrast to the fast Fourier transform, which has a computational complexity $\text{poly}(N\log_2(N))$. Since $N=2^n$, the fast Fourier transform is exponential in the size of the input, whereas the quantum Fourier transform is polynomial in the size of the input, as far as computational time is considered.

At this point one may wonder why the fast Fourier transform, with its exponential time complexity, is called *fast*. The reason is that, although strictly speaking the size of the input is given by the number of bits $n=\log_2(N)$, in many applications the size of input does not grow that fast. Instead, it is N, which is the size of the input, making the fast Fourier transform almost linear in N, which is an enormous improvement over the N^2 complexity of the matrix-multiplication-based Fourier transform.

3.6.5 Application of the quantum Fourier transform

Even though a quantum computer is able to efficiently perform a discrete Fourier transform this is not immediately useful. The Fourier transform is contained in the quantum register as a superposition of states. Upon observation of the quantum register containing the Fourier transformed function, the superposition collapses to a single component out of the superposition. The merit of the quantum Fourier transform is that it can perform the discrete Fourier transform for all components in parallel.

There exist, however, situations in which it is possible to take advantage of the Fourier transform by making use of specific properties of the Fourier transform. One such a property is that the discrete Fourier transform of a function f_j, with period r, is a function $\bar{f}_k$ which only is nonzero for values $k = m\frac{N}{r}$ with $m = 0, 1, \ldots, r-1$, and where N is the length of the transform. This property, which is proven in appendix A.2, can be utilized to *efficiently* determine the period r of the original function.

Classically there is no better algorithm for determining the period than by trial and error: calculate the function for arbitrary values of the argument, if two identical values are obtained then these are one or more periods apart. Repeating this proces allows to obtain the period in $O(N)$ function evaluations. On the other hand, on a quantum computer, exploiting quantum parallelism and the quantum Fourier transform, the period can be obtained in one function evaluations plus $O(\log^2(N)$ operations for the Fourier transform.

The algorithm that determines the period of a function is as follows. A quantum computer consisting of $2n$ qubits is supposed to be at disposal. The first n qubits make up the input register and the last n qubits make up the output register. Initially both the input, and the output register are set to zero. The state of the entire register will be denoted ψ, with an appropriate subscript. The initial state denoted ψ_0 is

$$\psi_0 = |0\rangle_1 |0\rangle_2 \dots |0\rangle_n |0\rangle_1 |0\rangle_2 \dots |0\rangle_n \tag{3.43}$$

$$= |0\rangle|0\rangle \tag{3.44}$$

where, again, the same notation is used for the state of a single qubit, $|0\rangle_i$ for the i-th qubit, and for the state $|0\rangle$ of a whole input or output register. In the next step the input register is prepared in a superposition of all possible inputs, i.e., a superposition of the numbers $0, 1, \dots 2^n - 1$. This can be done by applying a Hadamard transformation on each of the individual qubits of the input register. As always is $2^n = N$. This gives the state ψ_{start} that reads

$$\psi_{start} = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} |j\rangle|0\rangle. \tag{3.45}$$

The next step is to evaluate the function $f(x)$, in parallel for all possible inputs, by an appropriate circuit of logic gates. Because the output register contains zero before the evaluation it will contain the function after the evaluation, not the function exclusive or:ed with its argument, see also section 3.5.3. Also, the output will be in the form of a superposition of function values for all values of the argument, in a so called entangled form. The entire register will now be in state ψ_{func} reading

$$\psi_{func} = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} |j\rangle|f(j)\rangle. \tag{3.46}$$

The function f has the unknown period r, meaning that $f(x) = f(x + r)$, which is assumed to be a divisor of N, i.e., $N/r = K$, with all of N, r, K, x, and f integer numbers. The summation over j in equation 3.46 can be partitioned into a summation over one

period and a summation over all periods, in a way similar to appendix A.2 giving.

$$\psi_{func} = \frac{1}{\sqrt{r}} \sum_{j=0}^{r-1} \frac{1}{\sqrt{K}} \sum_{q=0}^{K-1} \frac{1}{\sqrt{K}} |j + qr\rangle |f(j)\rangle \tag{3.47}$$

Note that the summation over q only affects the input register, since the function in the output register does not contain q because of the periodicity $f(j+qr) = f(j)$. The next step is to observe the output register, returning a single function value out of the superposition, and leaving this function value in the output register. Because of the periodicity of f there are, at most, r different function values possible. The input register was prepared such that all input values occurred with equal amplitude in the superposition, therefore each of the r function values $f(j)$ have equal probability of being the result of the observation. Say one obtains $f(j_0)$, then after observation the state of the entire register is

$$\psi_{obs} = \frac{1}{\sqrt{K}} \sum_{q=0}^{K-1} \frac{1}{\sqrt{K}} |j_0 + qr\rangle |f(j_0)\rangle. \tag{3.48}$$

The input register still contains a superposition of all values of the argument that produce $f(j_0)$. Measuring the input register will return $j_0 + qr$ for some, unknown value of q, which does not give information about the period r.

At this point the quantum Fourier transform is applied to the input register only, producing

$$\mathrm{DFT}_N : \psi_{obs} = \frac{1}{\sqrt{K}} \sum_{q=0}^{K-1} \Big[\mathrm{DFT}_N : |j_0 + qr\rangle \Big] |f(j_0)\rangle \tag{3.49}$$

$$= \frac{1}{\sqrt{K}} \sum_{q=0}^{K-1} \Big[\frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{(j_0+qr)\frac{2\pi i}{N}k} |k\rangle \Big] |f(j_0)\rangle \tag{3.50}$$

$$= \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} \Big[\frac{1}{\sqrt{K}} \sum_{q=0}^{K-1} e^{qr\frac{2\pi i}{N}k} \Big] e^{j_0\frac{2\pi i}{N}k} |k\rangle |f(j_0)\rangle. \tag{3.51}$$

The factor in square brackets in equation 3.51 can be evaluated in a way similar to as is done in appendix A.2.

$$\frac{1}{\sqrt{K}} \sum_{q=0}^{K-1} e^{qr\frac{2\pi i}{N}k} = \frac{1}{\sqrt{K}} \sum_{q=0}^{K-1} e^{q\frac{2\pi i}{K}k} \tag{3.52}$$

$$= \begin{cases} \sqrt{K} & k = mK, m = 0, 1, \ldots r-1 \\ 0 & k \neq mK \end{cases} \tag{3.53}$$

The result in equation 3.53 allows to reduce the sum over k in equation 3.49 to include only the nonzero terms $k = mK$, giving

$$\mathrm{DFT}_N : \psi_{obs} = \frac{1}{\sqrt{r}} \sum_{m=0}^{r-1} e^{j_0\frac{2\pi i}{N}mK} |mK\rangle |f(j_0)\rangle \tag{3.54}$$

$$= \frac{1}{\sqrt{r}} \sum_{m=0}^{r-1} e^{j_0\frac{2\pi i}{r}m} |mK\rangle |f(j_0)\rangle \tag{3.55}$$

If now the input register is measured, a value $m = \lambda K = \lambda \frac{N}{r}$ is returned, for some, unknown, value of λ, in the range $0 \ldots r-1$. In other words, one obtains some multiple of $\frac{N}{r}$, without knowing exactly what multiple. The numbers m and N are known, and λ and

r are unknown, and there is one equation connecting these numbers. Hence, in general one cannot solve r form this equation, but under certain circumstances one actually can solve for r.

In particular, if $\gcd(\lambda, r) = 1$, and writing

$$\frac{\lambda}{r} = \frac{m}{N} \tag{3.56}$$

one can obtain both λ, and r by cancelling all common factors of m, and N, after which λ equals the numerator of the result, and r the denominator of the result.

As a small example, suppose $N = 16$, the unknown period $r = 4$, the observation of the input register resulted in a multiple $\lambda = 3$, meaning $m = 12$ was obtained. These values satisfy $m = \lambda \frac{N}{r}$, and $\gcd(\lambda, r) = 1$. Cancelling common factors in $\frac{m}{N}$ produces

$$\frac{m}{N} = \frac{12}{16} = \frac{3}{4} \tag{3.57}$$

after which no further reduction is possible, and one can correctly deduce that $\lambda = 3$, and $r = 4$. The problem is that it is not possible to know in advance whether or not $\gcd(\lambda, r) = 1$, and if this condition is not fulfilled one will cancel common factors of λ, and r as well, resulting in a too small value for r.

The saving grace is that the *probability* that $\gcd(\lambda, r) = 1$ is rather high, actually at least as high as $1/\log(N)$. This implies that, on average, repeating the whole procedure $\log(N)$ times will, with certainty, produce a correct value for the period r.

The fact that the probability that $\gcd(\lambda, r) = 1$ is larger than $1/\log(N)$ can be deduced from the prime number theorem. If $\pi(x)$ denotes the number of primes less than or equal to x, then the prime number theorem states that

$$\lim_{x \to \infty} \frac{\pi(x)}{x/\log(x)} = 1. \tag{3.58}$$

In words the prime number theorem says that for large enough x the number of prime

numbers less than, or equal to x is $\frac{x}{\log(x)}$. Thus, on picking the value λ arbitrarily in the range $0 \ldots r-1$ the probability that λ is prime is $\frac{1}{\log(r)}$. Because the set of primes less than r is a subset of the set of numbers less than r that is coprime to r the probability that $\gcd(\lambda, r) = 1$, meaning that λ is coprime to r, is larger than $\frac{1}{\log(r)}$. Also, because $N > r$, and $\frac{1}{\log(r)}$ is a decreasing function the probability that an arbitrary number λ in the range $0 \ldots r-1$ is coprime to r is larger than $\frac{1}{\log(r)}$.

The assumption that r divides N exactly is of course not realistic. This assumption has been made to simplify the description of the whole procedure. This assumption is not crucial though, because for large enough N the spoiling effects of r not dividing N are small enough to not affect the procedure, we will not discuss this further.

In summary, the period of a function can be determined in $O(\log^3(N))$ operations. The quantum Fourier transform requires $O(\log^2(N))$ operations, and the whole procedure has to be repeated $O(\log(N))$ times resulting in an overall $O(\log^3(N))$ complexity.

3.6.6 Factoring on a quantum computer

An algorithm for the efficient determination of the period of a function has an important application: the integer factoring problem can be reduced to determining the period of a specific integer function. Given w, consider the equation

$$v^2 \equiv 1 \mod w \tag{3.59}$$

in the unknown v. If w is prime the only, trivial, solutions are $v = 1 \mod w$, and $v = -1 \mod w$. This can be seen by rewriting the equation as

$$(v-1)(v+1) = 0 \mod w \text{ or } (v-1)(v+1) = kw \tag{3.60}$$

If w is composite, $w = pq$, with p, and q prime numbers, then also pairs of non-trivial solutions exist. This can be clarified in the following way. Consider the equations

$$\begin{cases} v_1 \equiv 1 \mod p \\ v_1 \equiv -1 \mod q \end{cases} \tag{3.61}$$

$$\begin{cases} v_2 \equiv -1 \mod p \\ v_2 \equiv 1 \mod q \end{cases} \tag{3.62}$$

v_1, and v_2 in the sets of equations 3.61 and 3.62 both satisfy equation 3.59 because, for example, $v_1 \equiv 1 \mod p$ implies $v_1^2 \equiv 1 \mod p$, which implies $v_1^2 \equiv 1 \mod pq$. A solution to each of these sets also is a solution to equation 3.59. Such a a solution can be constructed with the help of the so called *Chinese remainder theorem*. For the case under consideration the theorem says that if $\gcd(p, q) = 1$ then the set of equations

$$\begin{cases} x \equiv a \mod p \\ x \equiv b \mod q \end{cases} \tag{3.63}$$

has a unique solution $x, 1 \leq x \leq pq$ given by

$$x \equiv (auq + bvp) \mod (pq) \tag{3.64}$$

where $uq \equiv 1 \mod p$, and $vp \equiv 1 \mod q$, saying that u is the modular inverse of q modulo p, and v is the modular inverse of p modulo q. In our case p and q are primes so that the condition $\gcd(p, q) = 1$ is fulfilled. From equations 3.64, 3.61, and 3.62 it follows that $v_1 = uq - vp \mod w$, and $v_2 = -uq + vp \mod w$. Writing $d = uq - vp$, it follows that $v_1 = d \mod w$, and $v_2 = -d \mod w$.

Now we have constructed the nontrivial solution d –the trivial solution also still exists–

to equation 3.59, thus d satisfies

$$(d-1)(d+1) = 0 \mod w \tag{3.65}$$

and $d \neq \pm 1$. Hence $(d-1)(d+1)$ is a multiple of w, or w divides $(d-1)(d+1)$. Because $d \pm 1 \mod w \neq 0$, it holds that $d \neq w-1$, and because $d \leq w$ it holds that $d \neq w+1$, therefore w must have a nontrivial factor in common with $d-1$, or with $d+1$. Therefore a factor of w can be found by calculating $\gcd(d+1, w)$ and $\gcd(d-1, w)$. The greatest common divisor of two numbers a and b can be calculated with Euclid's algorithm, which has a time complexity $O((\log(a))^2)$ [17], where a is the larger of the two numbers.

Now it has been shown that it is possible to factor the number w, provided one can set up an equation of the form 3.59. Given w, a v satisfying equation 3.59 is needed, after which a factor of w may be found by calculating $\gcd(v-1, w)$, and $\gcd(v-1, w)$. One may also obtain a trivial solution from which no factor of w can be deduced.

It remains to show that such an equation can be set up, as follows. The starting point is Eulers theorem which asserts that if n and m are relatively prime then

$$m^{\phi(n)} \equiv 1 \mod n \tag{3.66}$$

where $\phi(n)$ is Eulers totient function that is defined as the number of positive integers less than n which are coprime to n. Furthermore is $\phi(1) = 1$ by definition. For prime numbers p one has $\phi(p) = p-1$. Equation 3.66 shows that if n and m are relatively prime then there exists a power of m that is equal to 1 modulo n. Equation 3.66 does not exclude the possibility that there are other powers of m with the same property, it only says that there exists at least one such a power. Now call r the least power of m that equals 1 modulo n, then r is the order of m modulo n. That, in turn, means that $m^s \mod n$, as a function of s, is periodic, and has period r.

If y is coprime to w then, with the aid of Eulers theorem, v in equation 3.59 can be

replaced by y^r, giving

$$y^r = 1 \mod w \tag{3.67}$$

or, provided r is even,

$$(y^{r/2} - 1)(y^{r/2} - 1) = 1 \mod w \tag{3.68}$$

One can choose y coprime to w. If the period also is known, a factor of w may now be found, as described above, by calculating $\gcd(y^{r/2} - 1, w)$, and $\gcd(y^{r/2} + 1, w)$. No classical algorithm that efficiently determines the period of a function is known, but this can be done efficiently on a quantum computer with the quantum Fourier transform procedure.

Two assumptions have been made in the description of the algorithm that finds a factor of w, namely that the period r is even, and that a non-trivial solution of the equation $y^r \equiv 1 \mod w$ is obtained for $y^{r/2}$. None of these assumptions is justified, because one may obtain an odd r or a trivial solution. This makes the algorithm a probabilistic algorithm. If the algorithm fails in case one of the assumptions turns out not to be justified, one picks a new y and starts all over. It can be shown[28] that the probability that for an arbitrary integer y satisfying $\gcd(y, w) = 1$, the probability that r is even and that at the same time a non-trivial solution for $y^{r/2}$ is obtained, is larger than or equal to 0.5.

3.7 A fast quantum search algorithm

The time complexity of algorithms that search a database depends much on the state of the database. There are basically two kinds of states: ordered and unordered, also called sorted and unsorted.

The optimal search algorithm for a an ordered database consisting of N items is the binary search algorithm. The time complexity of this algorithm is $O(\log_2(N))$, hence

searching can be done fast. The time complexity of sorting the database, on the other hand, is, at best, $O(N\log_2(N))$.

In the case of an unordered database the only known algorithm for finding a certain item is to examine all items until the requested item is found. On average, half of the items will have to be examined, leading to a time complexity of $O(N)$.

It is assumed that the database consists of $N = 2^n$ items, this is not a restriction but simplifies the analysis. The process of looking up an item can be modelled by a function $f(x)$ that returns zero for all, but one, values of the argument,

$$f(x) = \begin{cases} 1 & x = x_0 \\ 0 & x \neq x_0 \end{cases} . \tag{3.69}$$

where x_0 plays the role of the searched for item.

Searching the database means that one is given a function, more or less like a black box, and one can evaluate the function for values x of the argument, continuing until the function returns 1 after which the value of the argument that yielded 1 is the requested item. The function is a black box, in the sense that one cannot examine the internals of the function and resolve the requested value x_0 that way. One can only calculate function values.

The idea of the quantum search is as follows. First the input register is prepared in a superposition of all possible inputs, all with equal coefficient, that is, amplitude. Quantum parallel evaluation of the function, though possible, does not lead anywhere since no way of extracting the result is known. Instead, a series of operations, $O(\sqrt{N})$ in number, is applied that affects the coefficients of the individual terms in the superposition in such a way as to relatively decrease the magnitude of the coefficients of $|x \neq x_0\rangle$ and, hence, relatively increase the magnitude of the coefficient of $|x_0\rangle$. It can then be shown that for N approaching infinity the coefficient of $|x_0\rangle$ will be 1, meaning that measurement of the register will, with certainty, return the requested value x_0. For finite N, this probability

will be smaller than, though close to 1.

The following discussion of the algorithm is inspired by reference [27] which in turn is based on the original paper [22] from 1996 by L.K. Grover, the discoverer of the algorithm. Our discussion is in certain respects more detailed though.

Grover introduced the following two unitary operations.

1. The operator I_{x_0} that inverts the amplitude of $|x_0\rangle$ and leaves all other states unaltered. The formal definition is

$$I_{x_0}|x\rangle = \begin{cases} |x\rangle & x \neq x_0 \\ -|x\rangle & x = x_0 \end{cases} \tag{3.70}$$

2. The operator

$$D = -H_n I_0 H_n \tag{3.71}$$

   where H_n is the direct product of n two by two Hadamard transformations $H \otimes H \otimes \cdots \otimes H$ and I_0 is I_{x_0} with $x_0 = 0$, i.e. the operator that inverts the amplitude of $|0\rangle_1 |0\rangle_2 \cdots |0\rangle_n$, and leaves all other computational basis states unaltered.

The operator I_{x_0} can be implemented in the following way. As usual a quantum register is partitioned into two parts, called input register and output register, but these are just names and do not restrict the use of these registers otherwise. In this case the input register consists of n qubits, and the output register consists of 1 qubit, which is in agreement with the function $f(x)$ returning a binary value.

In section 3.5.4 and appendix A.1 it was explained how the input state $|x\rangle$ could be transformed into $(-1)^{f(x)}|x\rangle$ through the operator U_f. Although the analysis applied to a single qubit the result is valid for n qubits as well. Since $f(x) = 1$ for $x = x_0$, and zero

otherwise, this procedure will exactly implement the operator I_{x_0}, see figure 3.16. With I_0, and H, now also D can be implemented.

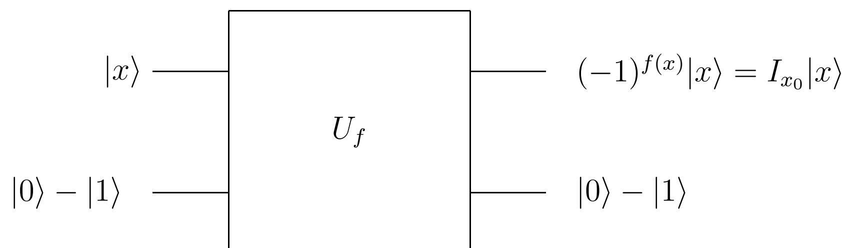


Figure 3.16: *Realization of I_{x_0} by means of U_f.*

Since the state of the output is unchanged through the whole procedure, as also seen from figure 3.16, it will be omitted from now on. The n qubits are first all prepared in the state $|0\rangle$, after which a Hadamard transformation is applied to each qubit in order to obtain a superposition with equal amplitudes, after which the state of the quantum register can be written as

$$|\psi_0\rangle = \frac{1}{N}\sum_{x=0}^{N-1}|x\rangle \tag{3.72}$$

The procedure of finding the value x_0 now consists of repeatedly applying the operator DI_{x_0} which generates the sequence of states $|\psi_1\rangle, |\psi_2\rangle, \ldots, |\psi_k\rangle, \ldots$, given by

$$|\psi_k\rangle = [DI_{x_0}]^k|\psi_0\rangle \tag{3.73}$$

which also can be written as a recurrence relation,

$$|\psi_{k+1}\rangle = DI_{x_0}|\psi_k\rangle \tag{3.74}$$

It now remains to show that these transformations increase the amplitude of $|x_0\rangle$ relative to all other amplitudes. To that end we first obtain a more explicit form for the matrix D.

The direct product of n Hadamard transformations acting on the state $|00\ldots0\rangle$ produces the sum of all possible states divided by the square root of N,

$$H_n|00\ldots0\rangle = H \otimes H \otimes \cdots \otimes H|00\ldots0\rangle = \frac{1}{\sqrt{N}}\sum_{x=0}^{N-1}|x\rangle \tag{3.75}$$

The crucial observation is that the operator H_n acting on an arbitrary state $|x\rangle$ will, apart from a pre-factor $\frac{1}{\sqrt{N}}$, produce some sum of all kinds of terms, always containing the term $|00\ldots0\rangle$, with only zeros, only once. Therefore will $\sqrt{N}H_n|x\rangle - |00\ldots0\rangle$ not contain the term $|00\ldots0\rangle$, and will be unaffected by I_0. Using this, one obtains,

$$\begin{aligned}
D|x\rangle &= -H_n I_0 H_n|x\rangle && (3.76)\\
&= -H_n I_0 \frac{1}{\sqrt{N}}\left[(\sqrt{N}H_n|x\rangle - |00\ldots0\rangle) + |00\ldots0\rangle\right] && (3.77)\\
&= -H_n \frac{1}{\sqrt{N}}\left[(\sqrt{N}H_n|x\rangle - |00\ldots0\rangle) - |00\ldots0\rangle\right] && (3.78)\\
&= -H_n \frac{1}{\sqrt{N}}\left[\sqrt{N}H_n|x\rangle - 2|00\ldots0\rangle\right] && (3.79)\\
&= -H_n H_n|x\rangle + 2\frac{1}{\sqrt{N}}H_n|00\ldots0\rangle && (3.80)\\
&= -I|x\rangle + \frac{2}{N}\sum_{y=0}^{N-1}|y\rangle && (3.81)
\end{aligned}$$

where also equation 3.75 has been used, as well as that H_n is its own inverse.

From definition 3.70 for I_0 it is clear that under the action of I_0 the amplitudes of all $|x\rangle$ with $x \neq x_0$ will remain equal to each other. The same applies to D, as can be inferred from the explicit form 3.81, and hence for their product.

The state $|\psi_k\rangle$ from 3.73 can therefore be written as

$$|\psi_k\rangle = \alpha_k \sum_{x\neq x_0} |x\rangle + \beta_k |x_0\rangle \tag{3.82}$$

The coefficients α_k, and β_k can be determined by deriving a recurrence relation using equation 3.74

$$\begin{aligned}
|\psi_{k+1}\rangle = \alpha_{k+1} \sum_{x\neq x_0} |x\rangle + \beta_{k+1}|x_0\rangle &= DI_{x_0}|\psi_k\rangle && (3.83)\\
&= DI_{x_0}\alpha_k \sum_{x\neq x_0} |x\rangle + \beta_k|x_0\rangle && (3.84)\\
&= \alpha_k \sum_{x\neq x_0} DI_{x_0}|x\rangle + \beta_k DI_{x_0}|x_0\rangle && (3.85)\\
&= \alpha_k \sum_{x\neq x_0} D|x\rangle - \beta_k D|x_0\rangle && (3.86)\\
&= \alpha_k \sum_{x\neq x_0} \left[-|x\rangle + \frac{2}{N}\sum_{y=0}^{N-1}|y\rangle\right] - \beta_k\left[-|x_0\rangle + \frac{2}{N}\sum_{z=0}^{N-1}|z\rangle\right] && (3.87)\\
&= -\alpha_k \sum_{x\neq x_0} |x\rangle + \alpha_k(N-1)\frac{2}{N}\sum_{y=0}^{N-1}|y\rangle + \beta_k|x_0\rangle - \beta_k\frac{2}{N}\sum_{z=0}^{N-1}|z\rangle && (3.88)\\
&= -\alpha_k \sum_{x\neq x_0} |x\rangle + \alpha_k\frac{2(N-1)}{N}\sum_{x\neq x_0}|x\rangle + \alpha_k\frac{2(N-1)}{N}|x_0\rangle + \beta_k|x_0\rangle && (3.89)\\
&\quad - \beta_k\frac{2}{N}|x_0\rangle - \beta_k\frac{2}{N}\sum_{x\neq x_0}|x\rangle && (3.90)\\
&= \left((1-\frac{2}{N})\alpha_k - \frac{2}{N}\beta_k\right)\sum_{x\neq x_0}|x\rangle + \left((1-\frac{2}{N})\beta_k + (N-1)\frac{2}{N}\alpha_k\right)|x_0\rangle && (3.91)
\end{aligned}$$

where it has been used that a double sum over a quantity with one variable to be summed over reduces to a single sum times the number of terms in the outer summation. From equations 3.83 and 3.91 the recurrence relations follow, and the initial values for

α_k and β_k follow from equation 3.72, giving, together with a normalization condition that follows directly from 3.82, leading to

$$\alpha_0 = \beta_0 = \frac{1}{\sqrt{N}} \tag{3.92}$$

$$\alpha_{k+1} = (1 - \frac{2}{N})\alpha_k - \frac{2}{N}\beta_k \tag{3.93}$$

$$\beta_{k+1} = (1 - \frac{2}{N})\beta_k + (N-1)\frac{2}{N}\alpha_k \tag{3.94}$$

$$\beta_k^2 + (N-1)\alpha_k^2 = 1 \tag{3.95}$$

3.7.1 Closed form expressions for α_k and β_k

Although α_k and β_k are completely determined by equations 3.92 through 3.95, it is, for further analysis, preferable to have closed form expressions for α_k and β_k. These can be obtained as follows. The normalization condition 3.95 shows that α_k and β_k are related. As a matter of fact they can both be expressed as a function of the single variable θ_k,

$$\alpha_k = \frac{1}{\sqrt{N-1}}\cos(\theta_k) \tag{3.96}$$

$$\beta_k = \sin(\theta_k) \tag{3.97}$$

which satisfy the normalization condition. Now the unknown θ_k has to be solved for. We will show that the solution is given by

$$\theta_k = (2k+1)\arcsin(\frac{1}{\sqrt{N}}) = (2k+1)\theta \tag{3.98}$$

where the variable θ, defined by $\sin(\theta) = \frac{1}{\sqrt{N}}$ has been introduced. Hence, the final

form for α_k and β_k is

$$\alpha_k = \frac{1}{\sqrt{N-1}} \cos((2k+1)\theta) \tag{3.99}$$

$$\beta_k = \sin((2k+1)\theta) \tag{3.100}$$

In order to show that equations 3.99 and 3.100 satisfy the recursion relations some elementary equalities are useful.

1. Using $\cos(x) = \sqrt{1-\sin^2(x)}$ with $x = \arcsin(y)$ it follows that $\cos(\arcsin(y)) = \sqrt{1-y^2}$ which implies, on $\arcsin(\frac{1}{\sqrt{N}}) = \theta$, that $\cos(\theta) = \sqrt{\frac{N-1}{N}}$.
2. From $\sin(2\theta) = 2\sin(\theta)\cos(\theta)$ it follows that $\sin(2\theta) = \frac{2}{N}\sqrt{N-1}$, and from $\cos(2\theta) = \cos^2(\theta) - \sin^2(\theta)$ it follows that $\cos(2\theta) = 1 - \frac{2}{N}$
3. The standard geometrical equalities
   (a) $\cos(\phi_1)\cos(\phi_2) - \sin(\phi_1)\sin(\phi_2) = \cos(\phi_1 + \phi_2)$
   (b) $\sin(\phi_1)\cos(\phi_2) + \cos(\phi_1)\sin(\phi_2) = \sin(\phi_1 + \phi_2)$

We shall now finally show that equations 3.99 and 3.100 are solutions to recursion relations 3.93 and 3.94 by deriving these recursion relations from the expressions for α_k, and β_k. From equation 3.99, substituting $k+1$ for k, one obtains

$$\alpha_{k+1} = \frac{1}{\sqrt{N-1}} \cos\left[(2k+1)\theta + 2\theta\right] \tag{3.101}$$

$$= \frac{1}{\sqrt{N-1}} \left\{\cos\left[(2k+1)\theta\right]\cos(2\theta) - \sin\left[(2k+1)\theta\right]\sin(2\theta)\right\} \tag{3.102}$$

$$= \frac{1}{\sqrt{N-1}} \left\{\cos\left[(2k+1)\theta\right](1-\frac{2}{N}) - \sin\left[(2k+1)\theta\right]\sqrt{N-1}\frac{2}{N}\right\} \tag{3.103}$$

$$= (1-\frac{2}{N})\alpha_k - \frac{2}{N}\beta_k \tag{3.104}$$

which is the recursion relation for α_k. Similarly from equation 3.100, substituting $k+1$ for k, one obtains

$$\beta_{k+1} = \sin\left[(2k+1)\theta + 2\theta\right] \tag{3.105}$$
$$= \sin\left[(2k+1)\theta\right]\cos(2\theta) + \cos\left[(2k+1)\theta\right]\sin(2\theta) \tag{3.106}$$
$$= \sin\left[(2k+1)\theta\right](1-\frac{2}{N}) + \cos\left[(2k+1)\theta\right]\sqrt{N-1}\frac{2}{N} \tag{3.107}$$
$$= (1-\frac{2}{N})\beta_k + (N-1)\frac{2}{N}\alpha_k \tag{3.108}$$

which is the recursion relation for β_k

3.7.2 Number of iterations

Having established the procedure in detail, the question is how many times the unitary operator $[DI_{x_0}]$ in equation 3.73 has to be applied. In other words, how many iterations are required, or for what value of k is the problem solved? The answer follows immediately from equation 3.82: the problem is solved when $\alpha_k = 0$, and $\beta_k = 1$, because then a measurement of the input register will with certainty return the value x_0. The values $\alpha_k = 0$ and $\beta_k = 1$ are assumed if $(2k = 1)\theta = \frac{\pi}{2}$. The variable θ is related to N through $\sin(\theta) = \frac{1}{\sqrt{N}}$ which for small values of θ, i.e., large values of N implies that $\theta = \frac{1}{\sqrt{N}}$. The sought value for k, for large values of N, is therefore

$$k = \frac{\pi}{4}\sqrt{N} - \frac{1}{2}. \tag{3.109}$$

One can conclude that the time complexity of Grover's algorithm for finding an item in an unordered database is $O(\sqrt{N})$, in contrast to the $O(N)$ complexity of a classical search.

The number of iterations k is of course a whole number, while the value $k = \frac{\pi}{4}\sqrt{N} - \frac{1}{2}$ normally not will be a whole number. Then the value for θ where β_k equals exactly 1 can

not be attained. By choosing the closest possible whole number value of k β_k will be close to 1, and α_k will be close to 0, and the value x_0 will be extracted with high probability. These considerations do not affect the time complexity $O(\sqrt{N})$ of the algorithm, since one has to repeat the whole procedure at most a small number of times, independent of the value of N.

3.8 Summary

Three algorithms have been discussed that would run on a quantum computer, if constructed. These algorithms solve certain problems with an efficiency much higher than the efficiency of the best known algorithms, solving the same problems, running on a conventional computer. Algorithms can be analyzed in different ways, for example by analyzing the Turing machine equivalent of the algorithm, as shown in chapter 2, or by analyzing the logical circuit that implements the algorithm, as done in this chapter.

Especially the efficient solution of the integer factoring problem on a quantum computer has consequences for public key cryptography, discussed in the next chapter.

Chapter 4

Cryptography and quantum key distribution

4.1 Introduction and terminology

This chapter takes up the key distribution problem in cryptography. The purpose of cryptography is to provide the means for hiding the contents of a message from non-authorized persons. The usual procedure is to use a known method to encrypt the message, but keeping secret an essential ingredient necessary to perform both the encryption and decryption. This secret ingredient is called the key.

There exist provably unbreakable, and hence 100% secure methods, using a single key for both encryption and decryption. Such methods require that both the receiver and the sender have access to this otherwise secret key. This immediately leads to the key distribution problem. Two parties need a secret way of communication in order to establish a secret key that is needed for establishing a secure way of communicating.

The catch 22[20] situation that arises in connection with the key distribution problem is solved by public key cryptography, also called asymmetric cryptography, where the keys for encryption and decryption differ. Moreover, only the decryption key is kept secret.

Implementations of public key cryptography are based upon mathematical operations that are difficult to invert. This can be considered a mathematical solution to the key distribution problem. We also describe an entirely different method based upon physical principles that provides 100% security.

We summarize some terminology. Original, not encrypted, and therefore generally understandable message text is called *plaintext* whereas encrypted text is called *ciphertext*. Furthermore one distinguishes between stream ciphers that process a message bit for bit, and block ciphers, that break a message in blocks of certain length which are processed in turn, and return ciphertext blocks of the same length. As a matter of fact, the majority of currently used ciphers consists of block ciphers.

4.2 The Vernam cipher

In 1917 G.S. Vernam[4] invented a cipher that in 1949 by C.E. Shannon[5, 6, 7] was proven to be perfectly secure. With perfectly secure is meant that the cipher as such is unbreakable. The Vernam cipher is a stream cipher and is also known as the one-time pad cipher. The principle of the cipher is best illustrated with the help of a small example.

Suppose one uses the English alphabet consisting of 26 letters, uppercase only, and the space, in total 27 characters. In order to encrypt the phrase 'ONE TIME PAD', which consists of 12 tokens, first each token is replaced by its number in the alphabet, i.e., a number in the range $0 \cdots 26$. Then the key is added and the result is taken modulo 27. The resulting number is the encrypted character in its numerical form. Because the resulting number also lies in the range $0 \cdots 26$, the encrypted text can either be kept in numerical form, or in textual form.

The procedure is illustrated in table 4.1. The key consists of a row of *truly random* numbers in the range $0 \cdots 26$, i.e., the same range as the letters in the alphabet. Hence, the key is made up from the same alphabet as the plain text message, and both has a

Plain text	O	N	E		T	I	M	E		P	A	D
Plain text numeric	14	13	4	26	19	8	12	4	26	15	0	3
Encryption Key	F	S	M	Y	B	W	Y	P	W	H	T	N
Encryption Key numeric	5	18	12	24	1	22	24	15	26	7	19	13
cipher text numeric	19	4	16	23	20	3	9	19	25	22	19	16
cipher text	T	E	Q	X	U	D	J	T	Z	W	T	Q
cipher text numeric	19	4	16	23	20	3	9	19	25	22	19	16
Decryption Key	F	S	M	Y	B	W	Y	P	W	H	T	N
Decryption Key numeric	5	18	12	24	1	22	24	15	26	7	19	13
Plain text numeric	14	13	4	26	19	8	12	4	26	15	0	3
Plain text	O	N	E		T	I	M	E		P	A	D

Table 4.1: *The Vernam cipher*

textual and a numerical form. This means that the key can be constructed by randomly picking characters from the same alphabet as the message is made up of. Then both the plain text and the key are translated into a numeric equivalent after which the encrypted text, also called cipher text, is calculated.

The reverse procedure of translating the cipher text back into the original plain text message is almost identical to the encryption procedure. The only difference is that the key now is subtracted from, modulo 27 in this example, instead of added to the corresponding character of the message. Note that the same key is used for encryption and decryption.

The operations of encryption and decryption can, equivalently, be done bit for bit by letting each encrypted bit be the result of the exclusive or operation (XOR) of the plain text bit and the key bit. The XOR operation, at the binary level, is the same as binary

Plain text	O	N	E		T	I	M	E		P	A	D
Encryption Key	F	S	M	Y	B	W	Y	P	W	H	T	N
cipher text	T	E	Q	X	U	D	J	T	Z	W	T	Q
Decryption Key	F	S	M	Y	B	W	Y	P	W	H	T	N
Plain text	O	N	E		T	I	M	E		P	A	D

Table 4.2: *The Vernam cipher, text only*

addition modulo 2. The encryption process then amounts to translating the plain text message in a bit pattern, then choosing a random bit pattern of the same length, after which the encrypted tex is obtained as the result of the exclusive or operation on these two bit patterns. The original plain text message is retrieved by again applying the exclusive or operation, this time on the cipher text, but using *the same key*. It can easily be verified that this procedure results in the original plain text message. In this way both the encryption, and the decryption process can be done by *bit by bit processing*, which makes the Vernam cipher a stream cipher.

4.2.1 The security of the Vernam cipher

Shannon proved that the Vernam cipher is unbreakable if the key satisfies the following conditions.

1. The key has at least the same length as the message.
2. The key is never reused.
3. The key is truly random.

The first and the second condition can relatively easy be fulfilled. Generating a truly random key is not as easy but is in principle possible if, for example, a random number generator based on truly random processes, such as radio active decay, is used.

If the key is never reused and also is truly random the Vernam cipher also is called a *one time pad*, for obvious reasons. Under the conditions stated above the Vernam cipher is *unconditionally secure*. This means that the occurrence of *patterns*, or more generally, *statistical dependencies*, in the plain text, do not affect the security of the encrypted message. The reason for this is the following.

A secret system provides perfect secrecy against against cipher-text only attacks if the message and its encrypted form are statistically unrelated, according to Shannon. Since the

key is truly random the message and the key (which has the same length) are statistically unrelated. The result of the XOR operation on the message and the key produces another truly random sequence of bits which henceforth also is statistically unrelated to the original message.

It should be stressed that the one time pad only provides unconditional security if the key has the same length as the plain text message. This requirements may result in very long keys and may therefore sometimes be an undesirable feature. It is however possible to generate pseudo random sequences of arbitrary length that are based on a so called seed of much shorter length. In that case the system does no longer provide unconditional security, but still may provide computational security, i.e., even though the system can in principle be defeated, the required resources are supposed to be unavailable to anyone.

There is however another problem which puts limits on the practical usability of the Vernam cipher and that is that both the sender and the receiver of the message must have access to the same key. This constitutes the so called key distribution problem, which will be discussed in 4.3. However, given two persons that have access to a common key that fulfills the requirements above the Vernam cipher is unbreakable.

4.3 The key distribution problem

4.3.1 Preliminaries

Data encryption lets two people, *Alice* and *Bob*, *secretly* communicate over an insecure communication channel. However, in order for *Bob* to be able to decrypt what *Alice* writes it is necessary to agree upon what cryptographic system to use and also agree upon the values of the parameters in the chosen cryptographic system. Hence *Alice* and *Bob* must communicate *before* they have established a secure way of communicating with each other. Most cryptographic systems are composed of an algorithm and a key. The algorithm is publicly known and the key is to be kept secret. It is a value for the key that *Alice* and

Bob have to agree upon. Since *Alice* and *Bob* have not yet established a secure way of communicating with each other they have to find another way of establishing a secret key, for example meet. The paradoxical situation where *Alice* and *Bob* already need a secure communication channel in order to establish a secure communication channel is called the *key distribution problem*. in the following we will describe two different solutions to the problem.

4.3.2 A solution based on mathematical principles

A solution to the key distribution problem was presented in 1976 by Diffie and Hellman.[15] Their solution builds upon the observation that *Alice* only has to be able to encrypt a message, she does not have to be able to decrypt the messages that she is sending to *Bob*. Hence, by using two different keys, one for encryption and another one for decryption, the key distribution problem can be solved if *Bob* makes the encryption key publicly known but keeps the decryption key secret. Now everybody, including *Alice* can send encrypted messages to *Bob*, that only *Bob* can decrypt.

Because the system contains two keys, one secret decryption key, called the private key, and one publicly known encryption key, called the public key, such a cryptographic system is called a public key cryptographic system. Sometimes these systems are also called asymmetric cryptographic systems, as opposed to symmetric key cryptographic systems.

The following analogy can be used to describe public key cryptography. *Bob* supplies everyone who wants to send secret messages to him with an *unlocked* padlock. *Bob* keeps the key for opening the padlocks, this may be the same key that fits in all padlocks. Now *Alice* can supply *Bob* with a secret message by writing it on paper and putting the paper into a box which she locks with the padlock, which can be done without any key because of the normal construction of a padlock. The locked box is sent to *Bob* by the ordinary postal system. Since only *Bob* has the key to the locks it is only he who can open the boxes and read the messages. In this analogy, the set of unlocked padlocks is the public

key, the key to the padlocks is the private key.

The public key cryptographic system requires two keys, moreover it may not be possible to derive the decryption key from the encryption key. It is not clear from the outset whether or not algorithms that have the required properties exist. It turns out, however, that such algorithms do exist. These algorithms all have the common property of making use of a mathematical operation which is very difficult to invert. The most widespread public key cryptographic system currently is the so called RSA[14] system[1]. Here the operation which is difficult to invert is the multiplication of two large, e.g. 150 digits, prime numbers. Obtaining the product by multiplying the two primes is easy, but finding the factors given the product is computationally infeasible.

4.3.3 The RSA cryptosystem

In this section the RSA public key cryptographic system will be summarized. Multiplying together two large, i.e., 200 digits, prime numbers is an operation that is difficult to invert, since the inverse operation amounts to factoring integers. The fastest known algorithm for factoring integers is the so called number field sieve [18], that has a complexity $e^{1.9+O(1)\sqrt[3]{\log(n)\times(\log(\log(n)))^2}}$, where n is the number to factor. This is a super-polynomial complexity, and a very fast growing function of the size of the input $\log(n)$, even though the function grows sub-exponential because of the cubic root in the exponential.

For given computational resources there will be a largest input size L, beyond which factoring becomes infeasible. By choosing a number b of a size that exceeds L by just 1 digit, the factoring of b constitutes an infeasible problem for the given resources. Currently numbers of sizes in the range 1024 to 2048 bits are considered impossible to factor using currently known methods and current technology.

The RSA cryptosystem incorporates the factoring problem via the totient function,

[1]The acronym RSA stands for Rivest Shamir Adleman, which are the names of the originators of the method

already discussed in section 3.6.6, that satisfies Euler's theorem, given in equation 3.66. If n is a prime number, Euler's theorem can be written in a slightly different form, see for example reference [8], reading

$$m^{k\phi(n)+1} \mod n = m \tag{4.1}$$

where $0 < m < n$, and k a positive integer. Equation 4.1 shows that the number m after modular exponentiation to the power $k\phi(n) + 1$, with modulus n, reproduces the number m. This reproduction is exactly what is required by an encryption followed by a decryption.

If both encryption and decryption are done by exponentiation modulo n, using different exponents, e for encryption, and d for decryption, then

$$C = M^e \mod n \tag{4.2}$$

$$M = C^d \mod n = M^{ed} \mod n \tag{4.3}$$

In other words, if

$$ed = k\phi(n) + 1 \tag{4.4}$$

the exponents e, and d are such that exponentiation modulo n with exponent e will encrypt a plain text message M, that can be decrypted by exponentiation modulo n with exponent d. In this way the key for decryption, consisting of the pair $\{d, n\}$ is different from the key for decryption, consisting of the pair $\{e, n\}$, yet does reproduce the original plain text message.

The secret decryption exponent d is related to the publicly known encryption exponent

e through equation 4.4, which mathematically expresses that e, and d are modular inverses, *with respect to the not publicly known modulus* $\phi(n)$. In principle $\phi(n)$ can be calculated, since it is a well defined function, and n is publicly known. There is however no known algorithm that efficiently calculates $\phi(n)$ for non-prime numbers n. In fact, the fastest known way to calculate $\phi(n)$ is to first factor n, in this case in the two prime factors p, and q, and then calculate $\phi(n)$ using that $\phi(pq) = (p-1)(q-1)$, for p and q prime. Since, as explained above, it is believed that factoring numbers of sufficiently large size is infeasible, it also is believed to be impossible to decrypt without knowing $\phi(n)$. However, a polynomial time algorithm for factoring integers on a quantum computer already exists, see chapter 3.

4.3.4 A solution based on physical principles

The above solution of the key distribution is based on mathematical principles. It is also possible, at least in principle, to arrive at a solution for the key distribution problem based on physical principles. This solution is based upon the fact that certain physical phenomena can be accurately described only by quantum mechanics. This especially applies to phenomena that happen on a small, i.e., atomic, length scale. A very important consequence of a quantum mechanical description of a phenomenon is that the outcome of measurements has a probabilistic character.

Historically, matter was attributed a particle-like character, whereas other phenomena, like light, were attributed a wave-like character. One of the revolutionary new ideas in quantum mechanics was that, on the one hand waves were needed for a proper description of particles, and, on the other hand, wave like phenomena like light, in certain respects showed a particle like behaviour. This dual behaviour is called the wave-particle duality.

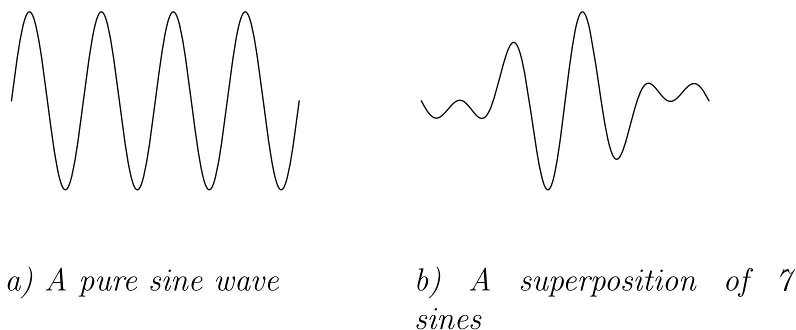
One of the consequences of the wave-like character of particles is uncertainty in either the position or the velocity of the particle. In a classical, i.e. non quantum mechanical description of a particle, the position and velocity of a particle are always fully determined.

One may not know either the position or the velocity, or both but then one can measure the unknown quantities.

A pure wave, however cannot be attributed a position. A wave is dislocated over, in principle, entire space. Hence, the location in space of a pure, sinus formed, wave is fully undetermined. The momentum of such a wave is, on the other hand, exactly determined, and is given by

$$p = \frac{\hbar}{\lambda}, \tag{4.5}$$

where λ is the wavelength, and $\hbar = 1.054610^{-34}$, which is Planck's constant divided by 2π . A superposition of a number of pure sinus waves behaves entirely different. In figure 4.1 a) a pure sine wave is plotted, and in figure 4.1 b) a superposition of 7 sinewaves that differ a small amount in wavelenght from the original wave is plotted. As a consequence of interference the resulting wave has different amplitudes in different areas. As a matter of fact the amplitude only has a significant value in a limited area and is almost zero elswhere. If the superpostion would consist of all possible wavelength, then the result would be a function that is zero everywhere, except at the origin, where it approaches infinity.



a) A pure sine wave *b) A superposition of 7 sines*

Figure 4.1: *Wave packet*

In this limiting case the resulting superposition is no longer distributed over entire space

but is, on the contrary, localized into a single point in space, like an idealized particle. However, this superposition of waves can no longer be attributed a unique momentum, since each component in the superposition has a different wavelength, and hence a different momentum.

A pure sine wave has fully determined momentum but completely undetermined position. A superposition of a number of waves with different wavelengths is no longer distributed over entire space, but the momentum of the superposition is no longer determined. The upshot is that by attributing a wave-like character to a particle it is no longer possible to simultaneously exactly determine position and momentum. Quantum mechanically this is expressed as

$$\Delta p \times \Delta x \geq \frac{\hbar}{2} \tag{4.6}$$

which is called Heisenberg's uncertainty relation, after the German physicist Werner Heisenberg. The symbol Δp stands for the uncertainty in momentum, and Δx for the uncertainty in position. The uncertainties Δx and and Δp can be given exact definitions in terms of expectation values which we will refrain from here. The name uncertainty relation is somewhat misleading, because it might suggest that there only exist uncertainty in the measurements and that one can determine both position and momentum simultaneously to arbitrary accuracy by refining the measurements. This is however not the case, the outcome of measurements of, for example, position is uncertain because the position is to a certain degree undetermined, and there is a certain probability to get a certain position as the outcome of the measurement.

Above it has been illustrated that as a consequence of the wave-particle duality measuring the position of a particle may give different results, even if the experiment is repeated under identical conditions. A probabilistic outcome of measurements is not limited to position but applies to all measurable quantities. Due to the extremely small size of Planck's constant, the probability distribution of the outcome of the measurements only deviates

significantly from what one would classically expect *for measurements on phenomena that take place on an atomic scale.*

The property that is used in quantum cryptography is the polarization plane of light. Light consists of electromagnetic waves. These waves, in turn, consist of an electric and a magnetic component. Because the direction of oscillation, for both the electric and the magnetic component, is at right angles to the direction of propagation, light is a so called *transverse* electromagnetic wave. Moreover, the electric and magnetic components oscillate in planes that are oriented at right angles with respect to each other, see figure 4.2.

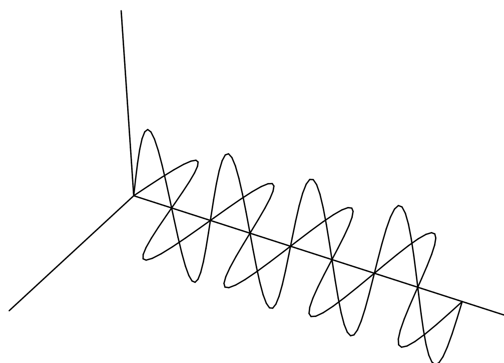


Figure 4.2: *Electromagnetic wave*

For our purposes it is sufficient to consider the electric component only. If the electric component always oscillates in the same direction, the light is called linearly polarized. In that case the electric component and the direction of propagation always form the same plane. Other polarizations also exist, for example circularly polarized light, where the directions in which the electric and magnetic components of the wave oscillate, rotate about the direction of propagation of the wave.

4.4 Quantum key distribution

Based on the principles described above, *Alice* and *Bob* can establish a secret key, i.e., a key only known to them, over an insecure communication channel. This procedure will involve two reference systems for the polarization of the photons. Given some reference direction, photons can be created with a polarization direction at an angle of either $0°$, or $90°$ with respect to the reference direction. Photons polarized in the $0°$ direction will encode a binary 0, and photons encoded in the $90°$ direction will encode a binary 1. The second reference system is rotated $45°$ with respect to the first system, meaning that the $45°$ direction also will encode a binary zero, but now in the rotated system, which will be called the diagonal system. Similarly a photon polarized in the $135°$ will encode a binary one in the diagonal system. The original, non-rotated system will be referred to as the rectilinear system.

A binary zero encoded in the rectilinear system will show up as a binary zero if it also is measured in the rectilinear system. If it is encoded in the rectilinear system but measured in the diagonal system there is a 50% probability of measuring a binary zero and a 50% probability of measuring a binary 1. Thus, if a random bit stream is encoded in either the rectilinear or the diagonal reference system, and the receiver does not know which of the system has been used, the receiver will obtain half of the bits correctly, but does not know which ones are correct. If the sent message is not random, the receiver might be able to recognize the correct bits and guess the rest of the message.

A protocol for establishing a key over a publicly accessible channel can be set up as follows. *Alice* randomly chooses the values for a number of bits. The key will be constructed from a subset out of these bits. For each chosen bit she also chooses at random a reference system to encode the bit in: either polarized in the rectilinear system, denoted as $\oplus$, or polarized in the diagonal system, denoted as $\otimes$. Furthermore, a binary zero in the rectilinear system is denoted as $-$, and a binary one as $|$. A binary zero in the diagonal system is denoted as $/$and a binary one as $\backslash$.

bit	1	1	0	0	1	0	1	0	0	0	0	1	0	0	0	0
Ref. sys.	$\otimes$	$\otimes$	$\otimes$	$\otimes$	$\oplus$	$\otimes$	$\otimes$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\otimes$	$\otimes$	$\oplus$	$\oplus$	$\otimes$
photon	$\backslash$	$\backslash$	$/$	$/$	$-$	$/$	$\backslash$	$\vert$	$\vert$	$\vert$	$\vert$	$\backslash$	$/$	$\vert$	$\vert$	$/$

Figure 4.3: *Encoding of bits*

Figure 4.3 exemplifies this step in the protocol. The first row contains 16 randomly chosen bits, the second row contains randomly chosen reference systems, and the third row contains the result of the encoding in the notation specified above.

Bob receives the polarized photons sent by *Alice* and randomly chooses a reference system for each photon, performs a measurement of the polarization which produces either a 0, or a 1. In those cases where *Bob* choses the same reference system as *Alice* had chosen to encode the bit *Bob* will obtain the correct value of the bit with 100% probability. If he chooses the wrong reference system there still is a 50% probability of obtaining the correct value for the bit, but, more importantly, there also is a 50% probability of obtaining the wrong value.

photon	$\backslash$	$\backslash$	$/$	$/$	$-$	$/$	$\backslash$	$\vert$	$\vert$	$\vert$	$\vert$	$\backslash$	$/$	$\vert$	$\vert$	$/$
Ref. sys.	$\oplus$	$\otimes$	$\otimes$	$\oplus$	$\otimes$	$\oplus$	$\otimes$	$\oplus$	$\oplus$	$\otimes$	$\otimes$	$\oplus$	$\otimes$	$\otimes$	$\oplus$	$\otimes$
obtained bit	1	1	0	0	0	1	1	0	0	0	1	0	0	0	0	0
send bit	1	1	0	0	1	0	1	0	0	0	0	1	0	0	0	0

Figure 4.4: *Decoding of bits*

Alice and *Bob* now proceed in two steps. Firstly they establish whether *Bob* has received the the stream of bits from *Alice* without being eavesdropped. If eavesdropping has taken place *Alice* and *Bob* end their communication, otherwise they proceed in establishing a common, secret key. Both these steps can be done over a classic, i.e. non-quantum, public, i.e., insecure, communication channel, because the bits from which the key will be constructed have already been transmitted.

The first step, detecting eavesdropping, is done as follows. *Alice* and *Bob* choose a

subset of the bits for which they tell each other what reference system each of them used. Moreover, *Alice* tells *Bob* the values of the bits that *Bob* ought to have observed for those cases where they used the same reference system. Without eavesdropping this would be unnecessary because then there is a 100% probability that *Bob* will observe the same bit value as *Alice* sent. An eavesdropper will however, after measuring, transmit the polarized photons to *Bob* but has to guess a polarization, i.e., a reference system. Out of the four possibilities $-$, $|$, /, \there is only one correct and the eavesdropper has a chance of 3 out of 4 to make the wrong choice. Therefore the probability of detecting eavesdropping as a function of the number t of tested bits is $1 - \left(\frac{3}{4}\right)^t$. If 16 bits are tested the probability of detecting eavesdropping is already 99%. By adding more bits one can obtain a probability arbitrary close to 1.

If *Alice* and *Bob* detect eavesdropping they will end their communication, otherwise they continue as follows. *Alice* tells *Bob* the reference systems she used for each bit. *Bob* compares *Alice*'s reference systems with the ones he used and tells *Alice* for which bits he used the same reference as *Alice*. In these cases, where the used the same reference system, i.e. polarization, they both know the value for that bit. Thus these bits form a sequence that only is known to *Alice* and *Bob*.

Note that it is the aspect of true randomness that is at the root of the above sketched protocol. Even if both channels, the quantum optical, and the conventional communication channel are eavesdropped this will not help the eavesdropper. *Alice* told *Bob* what he should have measured, but the eavesdropper does not know what he actually measured. Neither can *Alice* prepare a message and determine in advance what *Bob* will measure. Thus *Bob* will always detect if eavesdropping has taken place.

It is some kind of strange coincidence that quantum technology at the same time may obsolete certain implementations of public key cryptography, namely those based upon the integer factoring problem, and provide for an entirely different solution of the key distribution problem. The RSA public key cryptographic system relies on the computational

infeasibility of factoring integers on a conventional computer. Using a perceived quantum computer, integers can be factored with polynomial time complexity, as shown in chapter 3, thereby breaking the RSA cryptographic system. However, as shown in this chapter, quantum technology allows two parties to establish a secret key, which subsequently can be used in a conventional, i.e., symmetric encryption system, thereby providing an alternative solution to the key distribution problem.

Chapter 5

Yet another direction: DNA computing

The digital electronic computer can exist because processes in nature follow physical laws in the form of mathematical prescriptions. A one to one correspondence between, on the one hand, physical processes, in the form of changes of electronic states in integrated circuits, and, on the other hand, mathematical operations forms the foundation of electronic computing. This correspondence allows for the electronic computation of functions. Interestingly, since the connection is bidirectional, electronic computers can subsequently simulate physical processes, through the computation of functions.

There also are other ways of mapping the computation of a mathematical function onto some process in nature, electronic devices do not comprise the only possible way. Two basic requirements need to be satisfied, firstly the possibility to represent, or store data, and secondly the possibility to act on and change the data, i.e., process the data. In 1994 L. Adleman[29] showed how an instance of the so called directed hamiltonian path problem, which is a NP-complete problem, could be solved by manipulating strings of DNA.

DNA, desoxyribonucleic acid, is the medium in which, or on which, the genetic code

of living organisms is stored. It is built up of four different bases, denoted A, G, C, and T, which are abbreviations for adenine, guanine, cytosine, and thymine, and which are collectively called nucleotides. Information can thus be stored in DNA using the four letter alphabet $\{A, T, C, G\}$.

The base pairs A, and T are complementary, as are the pairs C, and G. A strand of DNA can form a double strand with a complementary strand of DNA, assuming the famous double helix structure. The sequence of bases in DNA can be transformed into another sequence by enzymes. Certain enzymes can cut a DNA strand at a specific location, which is specified by some specific, usually short, sequence of bases. Other enzymes, called ligates, can bond together two strands of DNA. Replication of DNA is possible with the use of a substance called polymerase, whereas exonuclease can destroy DNA.

The possibility to encode information in strands of DNA, together with bio-operations of the type listed above allow, at least in principle, to perform certain computations. The feasibility of such computations was shown, for the first time, in reference [29], by solving the hamiltonian path problem for the graph shown in figure 5.1 by means of biological operations on DNA molecules. A hamiltonian path is a path that starts at a given vertex v_{in}, ends at a given vertex v_{out}, and enters every other vertex only once. The biological operations mentioned above were used to implement the following algorithm, that solves the hamiltonian path algorithm, for a graph of n vertices.

1. Generate random paths through the graph.
2. Keep only those paths that begin at v_{in}, and end at v_{out}.
3. Keep only those paths that visit exactly n vertices.
4. Keep only those paths that enter all of the vertices of the graph at least once.
5. If any paths remain say “Yes”, otherwise say “No”.

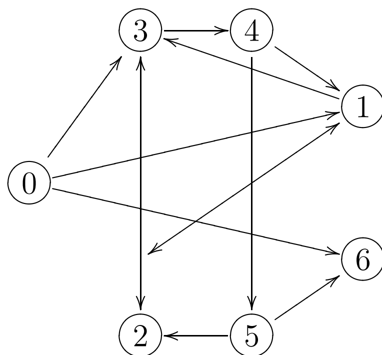


Figure 5.1: *Graph of seven vertices*

Step 1 was implemented as follows. Vertices were encoded in 20 nucleotide long strands of DNA, and oriented edges were encoded by bonding together two vertex-encoding 20 nucleotide strands. Subsequently, sequences of arbitrary length, containing compatible edges in arbitrary order, were formed. Edges are compatible if the target vertex j of the first edge (i, j) is the same as the source vertex of the second edge (j, k).

Step 2 was implemented by amplification of those molecules that started with $v_{in} = 0$ and ended with $v_{out} = 6$.

Step 3 was implemented by sorting the DNA strands by length, which separates out molecules consisting of the right number of vertices. This was done by gel-electrophoresis, a process where molecules travel through a wet gel, and where longer molecules travel more slowly than shorter ones.

Step 4 was implemented by means of affine purification, a process that filters out molecules containing a given sequence of nucleotides. By repeating the process all molecules containing each vertex at least once could be filtered out.

Step 5 was implemented by amplification of the result of step 4 followed by the determination of the DNA sequence of the amplified molecules.

The algorithm has a time complexity $O(n)$, where n is the number of vertices in the graph. On a classical, electronic computer, the problem belongs to the class of NP-complete

problems, hence the amount of time to solve the problem increases faster than any polynomial in n. Thus, Adleman's experiment not only showed that biological computation is possible, but also that in this way problems might be tackled that are intractable on an electronic computer.

The underlying reason for the efficiency of the algorithm above is *massive parallelism*, all molecules in the solution are processed simultaneously. The time complexity is determined by those steps that cannot be done in parallel, namely building up molecules that represent paths in the graph, where for n vertices there are n steps in the process.

Even though the generation of all possible paths is done in parallel, it is done, and each one requires its own small DNA computer. Hence, instead of being exponential in time the algorithm is exponential in the number of DNA-based processors, and therefore exponential in the mass of the amount of DNA. It is estimated that the amount of DNA needed to solve a 200 vertex hamiltonian path problem would have a mass exceeding the mass of the earth.

Two important questions concerning DNA computing are:

i) What kind of problems can be solved by DNA computing?

ii) Is it possible to design a *programmable* DNA computer?

In order to answer these questions a model of biological computation is needed. There exists a model called the splicing system model, in which the answers to these questions turn out to be affirmative[31]. These matters will not be pursued any further here.

The experiment done by Adleman really was a demonstration that DNA computing is in principle possible, but there are still many experimental problems, even reproducing Adlemans demonstration is difficult. At this point it is very difficult to speculate in what the possible applications, if any, of DNA techniques will be.

Chapter 6

Conclusion and summary

A number of aspects of computing have been considered, starting with the most important concepts from theoretical computer science. It has been shown that certain problems have no algorithmic solution. An example of such a problem is the general problem of software verification.

There exist many problems that have a time complexity such that, for most inputs, it is practically impossible to solve these problems on todays electronic computers. Some of these problems can be efficiently computed on a quantum computer or a DNA computer.

No algorithm is known that factors composite integers in polynomial time, a fact that leads to the classification of this problem as intractable. It is not known whether or not an efficient solution to the integer factoring problem exists at all.

However, it has been theoretically shown that quantum mechanical phenomena allow for the possibility of simultaneously processing huge amounts of information, using an effect called quantum parallelism. In this way a polynomial time solution to the integer factoring problem could be obtained if a quantum computer can be built.

This would challenge the RSA public key cryptographic system, which relies entirely on the presumed computational intractability of the integer factoring problem. Quantum optical technology, however, also is capable of providing a 100% secure solution to the key

distribution problem.

Another intractable problem, the hamiltonian path problem, also has been attacked from an entirely new direction, called DNA computing. DNA molecules in a solution containing enzymes are all processed at the same time. Using this fact together with the possibility to encode information in DNA strands, as with genetic information, it is possible to achieve massively parallel processing of information. It has been demonstrated that this procedure works by solving a small instance of the hamiltonian path problem.

Although these new technologies in principle have a large potential it is currently not clear whether they ever will be practicable. Furthermore, it is not known how generally applicable these technologies are. Their usability may be limited to very specific applications.

References

[1] A.M. Turing, “On computable numbers with an application to the Entscheidungsproblem”, Proc. London Math. Society, Vol. 2:42, pp.230-265, 1936. Ibid. 2:43, pp.544-546.

[2] Yury Matiyasevich, “Enumeable sets are diophantine”, Doklady Akademii Nauk SSSR,191 (1970), 279-282. English translation with addendum, Soviet Math.: Doklady, 11 (1970), 354-357.

[3] J. P. Jones, D. Sato, H. Wada, D. Wiens, “Diophantine representation of the set of prime numbers”, The American Mathematical Monthly, 83(6):449-464,1976.

[4] G.S. Vernam, “Cipher printing telegraph systems for secret wire and radio telegraphic communications”, Journal of the American Institute for Electrical Engineers, Vol. 55, pp. 109-115, 1926.

[5] C.E. Shannon, “A mathematical theory of communication”, Bell System Technical Journal, Vol. 27, Oct 1948, pp 379-423, 623-656.

[6] C.E. Shannon, “Communication Theory of Secrecy Systems”, Bell System Technical Journal, Vol 28, Oct 1949, pp 656-715.

[7] C.E. Shannon, “Prediction and Entropy of printed English”, Bell System Technical Journal, Vol 30, Jan 1951, pp 50-64.

[8] W. Stallings, “Cryptography and Network Security”, Prentice Hall, 1998.

[9] P. Williams, Scott H. Clearwater, “Explorations in Quantum Computing”, Springer Verlag, New York, 1998.

[10] A.J. Menezes, P.C. van Oorschot, S.A. Vanstone, “Handbook of applied cryptography”, Alfred J. Menezes, Paul C. van Oorschot, Scott A. Vanstone CRC Press LLC, 1997.

[11] B. Schneier, “Applied Cryptography”, John Wiley & Sons, Inc. , 1996, 2nd. ed.

[12] P. Garret, “Making Breaking Codes”, Prentice Hall, 2001.

[13] Ch. Kaufman, R. Perlman, M. Speciner, “Network security”, , Prentice Hall, 1995.

[14] R. Rivest, A. Shamir, L. Adleman, “A method for Obtaining Digital Signatures and Public Key Cryptosystems”, Communications of the ACM,February 1978.

[15] W. Diffie, M. Hellman, “New directions in Cryptography”, IEEE Transactions on Information Theory, November 1976.

[16] T. ElGamal, “A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms”, IEEE Transactions on Information Theory, vol. IT-31(4), pp.469-472, July 1985.

[17] D.E. Knuth, “The Art of Computer programming” Vol. 2, Addison-Wesley, 1981, 2nd ed.

[18] A. K. Lenstra, H. W. Lenstra, Jr., M. S. Manasse, and J. M. Pollard, “The number field sieve”, Proc. 22$^{\mathrm{nd}}$ACM Symp. Theory of Computing (1990), 564-572.

[19] C. Pomerance, “A tale of two sieves”, Notices of the American Mathematical Society, 43, (1996), No 12, 1473-1485.

[20] J. Heller, “Catch 22”, 1961. In this book Heller describes the impossible situation of pilots in the second world war where the fear for flying and possibly not returning made them crazy. A pilot that applied for not having to fly anymore because the flying made him crazy got the answer that, referring to regulation 22, ’ ..if you don’t want to fly because flying makes you crazy then you are not crazy and you keep flying.’ The expression of a *catch 22 situation* has been incorporated into the english language to describe paradoxical situations.

[21] P. Shor, “Algorithms for quantum computation: Discrete Logarithms and Factoring”, *Proceedings 35th Annual Symposium on Foundations of Computer Science* (1994), pp. 124-134.

[22] L. Grover, “A fast Quantum Mechanical Algorithm for Database Search”, *Proceedings of the 28th Annual ACM Symposium on the Theory of Computing (1996), pp. 212-219*

[23] R. Landauer, “Irreversibility and heat generation in the computing process”, IBM J. Res. Dev., 5:183, 1961.

[24] E. Fredkin, T. Toffoli, “Conservative Logic”, Int. J. Theor. Phys. , 21 (1982) 219-253.

[25] D. Deutsch, “Quantum Theory, The Church Turing Principle, and the Universal Quantum Computer”, Proc. Royal Soc. London, A400 (1985) 97-117.

[26] M.A. Nielsen, I.L. Chuang, “Quantum Computation and Quantum Information”, Cambridge University Press, 2000.

[27] D. Bouwmeester, A. Ekert, A. Zeilinger, “The Physics of Quantum Information”, Springer, 2000.

[28] A. Ekert, R. Jozsa, “Quantum computation and Shor’s factoring algorithm”, Rev. Mod. Phys. 68 (1996) 733

[29] L. Adleman, “Molecular computation of solutions to combinatorial problems”, Science, v.266, Nov.1994, 1021-1024.

[30] F. De Martini, V. Buzek, F. Sciarrino, C. Sias, “Experimental realization of the quantum universal NOT gate”, Nature, 419, october 24, 815-818 (2002).

[31] L. Kari, “DNA computing: the arrival of biological mathematics”, The mathematical intelligencer, 19, 2 (1997) 9-22

[32] Y. Bar-Hillel, M. Perles, E. Shamir, “On formal properties of simple phase-structure grammars”, Z. Phonetik. Sprachwiss. Kommunikationsforsch. **14** (1961), pp. 143-172.

Appendix A

A.1 Some details on Deutsch algorithm for parallel function evaluation

After being processed by the inverter and Hadamard gates, but before the processing by U_f, the state of the two-qubit system is

$$(\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)(\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = \frac{1}{2}\sum_{x=0}^{1}|x\rangle(|0\rangle - |1\rangle) \tag{A.1}$$

In order to obtain the effect of U_f on the right hand side of equation A.1 use the definition

$$U_f : |x\rangle|y\rangle = |x\rangle|x \oplus f(x)\rangle \tag{A.2}$$

and momentarily omit the summation over x. Then

$$U_f : |x\rangle(0\rangle - |1\rangle) = |x\rangle(|0 \oplus f(x)\rangle - |1 \oplus f(x)\rangle) \tag{A.3}$$

$$= \begin{cases} |x\rangle|0\rangle - |x\rangle|1\rangle & f(x) = 0 \\ |x\rangle|1\rangle - |x\rangle|0\rangle & f(x) = 1 \end{cases} \tag{A.4}$$

$$= \begin{cases} |x\rangle(|0\rangle - |1\rangle) & f(x) = 0 \\ -|x\rangle(|0\rangle - |1\rangle) & f(x) = 1 \end{cases} \tag{A.5}$$

$$= (-1)^{f(x)}|x\rangle(|0\rangle - |1\rangle) \tag{A.6}$$

Now add the summation over x and normalization factors giving,

$$U_f \frac{1}{2}\sum_{x=0}^{1} |x\rangle(|0\rangle - |1\rangle) = \frac{1}{2}\sum_{x=0}^{1}(-1)^{f(x)}|x\rangle(|0\rangle - |1\rangle) \tag{A.7}$$

Note that the factor $(-1)^{f(x)}$ cannot be put in front of the summation because the value of $f(x)$ depends, naturally, on x.

A.2 The discrete Fourier transform of a periodic function

In this section we prove a general property of periodic functions in relation with the discrete Fourier transform. Consider a discretized function $f_j, k = 0 \ldots N-1$. The discrete Fourier transform $\bar{f}_j, j = 0 \ldots N-1$ is defined as

$$\bar{f}_j = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{j\frac{2\pi i}{N}k} f_k \tag{A.8}$$

The form of the transform is such that the transformed function $\bar{f}$ always is periodic

with period N, which is a consequence of the the fact that the original function f contains N function values. Now consider a function that is periodic with period $r < N$, i.e., $f_{k+r} = f_k$. When computing the discrete Fourier transform one consider a whole number, say K of such periods, meaning that $N = Kr$, meaning that $N = Kr$.

The sum over k in equation A.8 can be rewritten as a sum ranging over one period, the first, and a sum over all linear transformations that map the first period on all the other periods. This means that the index k is rewritten as $k = p + qr$, where p ranges from 0 to $r - 1$, and q ranges from 0 to $K - 1$. For the function f it then holds that $f_{p+qr} = f_p$. Equation A.8 can now be rewritten as

$$\bar{f}_j = \frac{1}{\sqrt{N}} \sum_{p=0}^{r-1} \sum_{q=0}^{K-1} e^{j\frac{2\pi i}{N}(p+qr)} f_{p+qr} \tag{A.9}$$

$$= \frac{1}{\sqrt{N}} \sum_{p=0}^{r-1} e^{j\frac{2\pi i}{N}p} \sum_{q=0}^{K-1} e^{j\frac{2\pi i}{N}qr} f_{p+qr} \tag{A.10}$$

$$= \frac{1}{\sqrt{N}} \sum_{p=0}^{r-1} e^{j\frac{2\pi i}{N}p} f_p \sum_{q=0}^{K-1} e^{j\frac{2\pi i}{N}qr} \tag{A.11}$$

The second summation in equation A.11 can be evaluated as

$$\sum_{q=0}^{K-1} e^{j\frac{2\pi i}{N}qr} = \sum_{q=0}^{K-1} e^{j\frac{2\pi i}{K}q} = \begin{cases} \sum_{q=0}^{K-1} e^0 = K & j = 0, K, 2K, \ldots \\ \frac{1-e^{j\frac{2\pi i}{K}K}}{1-e^{j\frac{2\pi i}{K}}} = 0 & j \neq 0, K, 2K, \ldots \end{cases} \tag{A.12}$$

where the closed form formula for the summation of a geometric series, i.e.,

$$1 + r + r^2 + r^3 + \cdots + r^{n-1} = \sum_{m=0}^{m-1} r^m = \frac{1 - r^n}{1 - r} \tag{A.13}$$

with $r = e^{j\frac{2\pi i}{K}}$, and which holds if $r \neq 1$, has been used. Equation A.12 expresses the

fact that the Fourier transformed function f_j is nonzero only for values $j = mK = m\frac{N}{r}$, with $m = 0, 1, 2, \ldots$.