# MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„Introduction to recursion theory and computational complexity“

verfasst von / submitted by

Padraig Paul Sweeney, BA BEd MA

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Education (MEd)

Wien, 2024/ Vienna 2024

| | |
|---|---|
| Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet: | UA 199 520 525 02 |
| Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet: | Masterstudium Lehramt Sek (AB)<br>Unterrichtsfach Mathematik<br>Unterichtsfach Psychologie und Philosophie |
| Betreut von / Supervisor: | Assoz. Prof. Vera Fischer, Privatdoz. PHD |

## Abstract

This thesis investigates two different limitations of computation based on computability theory and computational complexity theory. On the one hand the theoretical boundary of computability, which results from the halting problem is described: as it turns out there are well defined functions, that are not computable. The first part of the thesis provides an overview of the main problems in recursion theory and a proof of the equivalence of two of the most famous approaches to computability namely the class of registermachine-computable functions and that of general recursive partial functions. On the other hand, practical limitations are investigated, that arise from the specific complexity of a problem respectively the algorithm to solve the problem. During the second part of the thesis deterministic and nondeterministic Turing machines are introduced and an overview of their connection to complexity theory is provided. In course of this, the „P=NP“-problem is presented.

## Zusammenfassung

Diese Arbeit untersucht anhand von Berechenbarkeitstheorie und Komplexitätstheorie zwei verschiedene Limitationen von Berechnungen. Einerseits geht es um theoretische Limitationen, die aus dem Halteproblem resultieren: es existieren wohldefinierte Funktionen, die nicht berechenbar sind. Der erste Teil der Arbeit bietet einen Überblick der Probleme der Berechenbarkeitstheorie und unter anderem einen Beweis der Äquivalenz von zwei der bekanntesten Arten Berechenbarkeit mathematisch in rigoroser Weise zu behandeln: Registermaschinen und allgemein partiell-rekursiven Funktionen. Andrerseits werden auch praktische Limitationen behandelt, die aus der Komplexität des Problems bzw. des Algorithmus zur Lösung des Problems resultieren. Der zweite Teil der Arbeit behandelt sowohl deterministische als auch nichtdeterministische Turingmaschinen und insbesondere deren Zusammenhang mit der Komplexitätstheorie. Im Zuge dessen wird auch das „P=NP“-Problem vorgestellt.

## Acknowledgement

First, I want to thank Vera Fischer for her support and introducing me to the topic of recursion theory and computational complexity. Furthermore, I would like to thank Kim and her parents Roya and Wolfgang for supporting me during my studies. Finally, I would like to thank my parents for their support. Both Kim´s parents and mine spent a lot of time with their grandsons Neil and Nolan to enable this thesis.

Contents

# 1. Introduction

Computability has practical and theoretical limits. Both areas are object of mathematical research. Theoretical boundaries result from limitations of calculability. Starting out with the informal concept of effectively calculable functions, the first mission of the thesis will be to demonstrate how two of the most famous concepts of formalizing effective calculablity, namely through general recursive functions and register machines, coincide. This seems very interesting because one would not assume this result just looking at the apparently very different definitions in use. But these are just two of several ways how to formalize the concept of effective calculabitlity. A list of famous equivalent concepts can be fund among other things in [1].

Even more Alonzo Church claimed in his "Church-Thesis", that no matter how one formalizes effective calculability, the result will always be equivalent to the concept of a computable partial function respectively a general recursive function, see [1]. A prude statement that is not set out to be proven, but - as will be shown later - has a lot of good evidence on its side. After briefly introducing the more informal concept of effectively calculable functions the thesis will go on to investigate the mathematically strict and formal concepts first of general recursive functions continuing with the concept of register machines.

Both of these concepts (perhaps all if you believe in the Church-Thesis and certainly all known) have a limitations. In both cases a Halting Problem occurs. As it turns out there are functions that can not be calculated even though they are well-defined, see[2] The analogy to Kurt Gödel's work on incompleteness, seeciteeh1, which proceeds Alan Turing's work about calculability a few years, will only be mentioned briefly at the end of the first part of this thesis.

Another possibility of formalizing effective calculability are Turing machines. The class of Turing-computable functions is equivalent to that of that of general recursive functions. The thesis will use the concept of Turing machines to demonstrate practical boundaries

of computations, that have a totally different origin. As it turns out there are spacial and timely limitation to what a deterministic Turing machine can calculate, see [3]. The crux is how fast the demand of spacial and timely resources grow compared to the amount of information the machine has to process. One of the most famous presently unsolved mathematical problems, namely the

$$\mathbb{P} = \mathbb{NP}$$

problem, concerns exactly this issues of time and space complexity. The term complexity refers to a strict mathematical concept of upper boundaries, one can assert required by a given computation.

The class $\mathbb{P}$ contains all the problems a deterministic Turing machine can solve in polynomial time respectively space. The class $\mathbb{NP}$ contains all the problems, that a nondeterministic Turing machine can solve in polynomial time respectively space. On the one hand it is easy to see that $\mathbb{P} \subseteq \mathbb{NP}$, see[3]. The difficult part is to decide, whether or not $\mathbb{NP} \subseteq \mathbb{P} \Rightarrow \mathbb{P} = \mathbb{NP}$ or if $\mathbb{P} \subsetneq \mathbb{NP} \Rightarrow \mathbb{P} \neq \mathbb{NP}$. The goal of the thesis is to lead give a definition of complexity classes and complexity hierarchy, in order to be able to demonstrate the nature of the $\mathbb{P} = \mathbb{NP}$ problem. [1]

## 2. Effective procedures

Trying to come up with an abstract way of describing algorithms the concept of effective calculable functions plays an important role. Effective calculability requires an effective procedure for a human or a program to carry out a finite number of well defined operations. After carrying out this operations the executed algorithm halts and delivers an answer. Take for example a very simple procedure to find out whether or not a given natural number is a prime number. You could for example try to divide the given number $n \in \mathbb{N}$ with every natural number smaller than $n$ greater than 1. If you can find a $m \in \mathbb{N} : m < n$ starting with 2, that is a divider of $m$ the algorithm tells you to halt and decide that $n$ is

[1]This problem is so difficult it even became one of the seven so called Millennium Problems the Clay Mathematics Institute in Cambridge published in the year 2000. [4]

not prime. If you can not find such an $m$ before you reach $n$, then you decide that $n$ is prime. (Of course there are a lot more efficient and faster ways to decide this).

Following this idea it is easy to see, that the set of prime numbers is a decidable set, because using the simple algorithm every member of the natural numbers can potentially be investigated respectively decided.

Important to the informal concept of effectively calculable functions is, that the amount of steps it takes to make a decision is finite. Naturally one will ask if every subset of the natural numbers is decidable. A short way to see, that this cannot be the case, is looking at the cardinality of the power set of the natural numbers $2^{\mathbb{N}}$ which is uncountable. Compare that to a list of every possible algorithm: this list is countable, because every algorithm only is allowed a finite amount of steps.

As mentioned above there are various ways to formalize this concept of effective calculability. The first concept this thesis will turn its attention to is the concept of general recursive functions. To begin with it will be necessary to give a few definitions to clarify the terminology used.

2.1. **Partial functions.** Partial functions are a generalization of functions. They come in handy whenever not all of the elements of a given domain have a corresponding element in the function's range. If a partial function is a total partial function, it is a function, meaning that every element of the functions domain is mapped of to the range of the function. Whenever we encounter domains of recursive functions which are not defined everywhere the concept of partial functions allows one to still investigate these structures in a mathematically meaningful way. Most of the definitions and theorems in the first part of the thesis are inspired by [6].

**Definition 1.** Let $D \subseteq \mathbb{N}^k$. A function $f : \mathrm{D} \to \mathbb{N}$ is said to be a partial function. If $D = \mathbb{N}^k$, we say that $f$ is a total function. Abusing notation, sometimes we will write $f : \mathbb{N}^k \to \mathbb{N}$, even if the function $f$ is not total.

**Definition 2.** A partial function $f : \mathbb{N}^k \to \mathbb{N}$ is called effectively calculable if the following is the case:

(1) If a $k$-tuple $\vec{x}$ is in the domain, then the procedure eventually halts end returns a finite value $f(\vec{x})$.
(2) If a k-tuple $\vec{x}$ is not in the domain of $f$, then the procedure does not halt at all.

**Definition 3.** The characteristic function of any set of $\mathbb{S} \subseteq \mathbb{N}^k$ is given by the piece wise function

$$C_s(\vec{x}) = \begin{cases} 1 & \text{if } \vec{x} \in \mathbb{S} \\ 0 & \text{if } \vec{x} \notin \mathbb{N}^k \setminus \mathbb{S} \end{cases}$$

The set $S$ is called decidable iff $C_s$ is a total partial function. If the domain of $C_s$ contains any undefined elements, then we call $S$ semidecidable:

$$C_s(\vec{x}) = \begin{cases} 1 & \text{if } \vec{x} \in \mathbb{S} \\ \uparrow & \text{if } \vec{x} \notin \mathbb{N}^k \setminus \mathbb{S} \end{cases}$$

As it turned out and will be shown in this thesis not every subset of $\mathbb{N}$ is decidable, resulting in $\mathbb{N}$ itself not being a decidable set. Nevertheless the natural number still remain a semidecidable set, although later property will not be discussed widely in the context of this thesis, there are several prominent theorems about semidecidable sets, e.g. Kleen's theorem, see[7] So now the first tools needed to talk about calculability in a mathematical meaningful way are packed: partial functions and decidability.

## 3. General Recursive Functions

The proofs and definitions in this section closely follow [1]. The class of general recursive functions is one way of formalizing the above mentioned informal concept of effective calculability: Starting out with primitive recursive functions and later adding the search

operator to obtain the concept of general recursive functions. David Hilbert even hypothesized in 1926, that every effectively calculable function could be built up from primitive recursive functions, see[8].

Later that year Wilhelm Ackermann - who was one of Hilbert's students- disproved Hilbert's hypothesis, by giving an example for a calculable function, that is not primitive recursive, see [5]. The Ackermann function eventually grows faster than any exponential function. But since this function isn't primitive recursive, another component is necessary to construct the Ackermann function and in particular to obtain the class of general recursive functions, which it lies in: the class containing all the functions that can be built up from primitive recursion and the search operator. The search operator will be defined and discussed later on in detail.

After introducing primitive recursive functions and giving a few examples of how well known operations are derived from it, the thesis will go on presenting and explaining the search- or $\mu$-operator. Most of the proofs and definitions are closely oriented on ideas from [1] and [6].

### 3.1. **Primitive Recursive Functions.**

**Definition 4.** A primitive recursive function $f : \mathbb{N}^k \to \mathbb{N}$ can be built from the following 5 items:

(1) The zero function $z : \mathbb{N}^k \to \mathbb{N}$ with $z(x) = (x_1, x_2, x_3, ..., x_n) = 0$

(2) The successor function $s : \mathbb{N} \to \mathbb{N} : s(x) = (x + 1)$

(3) The projection function $I : \mathbb{N}^k \to \mathbb{N}$ with $I_n^k(x_1, x_2, ..., x_n) = x_n$ , where $1 \leq n \leq k$

(4) Using composition $h(x) = f(g(x_1), g(x_2), ..., g(x_n))$

(5) And primitive recursion defined as follows, for the functions $h : \mathbb{N}^{k+1} \to \mathbb{N}$ , $g : \mathbb{N}^{k+2} \to \mathbb{N}$ and $f : \mathbb{N}^k \to \mathbb{N}$

(a) $h(x, 0) = f(x)$

(b) $h(x, y + 1) = g((h(x, y), x, y)$

So a primitive recursive functions is built up by four simple building stones: First of all the $0 - function$, composition and the successor function, which are all rather self explaining. Second of all a projection function, that allows us, to map an arbitrary $i - th$ component of a $k$-place element onto exactly the component. All three functions are total functions. Last but no least, we can use primitive recursion which allows one to fully describe a value by its predecessor. Primitive recursive functions are closed under composition and primitive recursion, meaning, that the class of primitive recursive functions is defined, as the smallest class that contains all primitive recursive functions and is closed under composition and primitive recursion. For a more extended explanation, see[1]

Now that we can be sure, that using all operations available will not enable us to leave the class of primitive recursive functions, it is interesting to look at some examples how well known functions can be built in this way. The two following examples are closely oriented on, [9]. The first example shows how to construct a function that adds two natural numbers by using the tool set of primitive recursive functions., the second example will demonstrate how addition can be used to define multiplication.

3.2. **Addition function.** The addition function $add : \langle a, b\rangle \to a + b$ can be defined using primitive recursion.

(1) $add(a, 0) = x$

(2) $add(a, b + 1) = h(a, b, add(a, b)) = S(I_3^3(a, b, add(a, b)$

3.3. **Multiplication function.** The multiplication function $m : \langle a, b\rangle \to a \times b$ can be defined using primitive recursion.

(1) $m(a, 0) = g(a) = z(a) = 0$

(2) $m(a, b + 1) = h(a, b, m(a, b))$

(3) $h(a,b,m(a,b)) = add(I_3^3(a,b,m(a,b)), I_1^3(a,b,m(a,b) = m(a,b) + a$

So we have constructed a program for multiplying two natural numbers, just using the Zerofunction, the projection function and primitive recursion and the primitive recursive addition function. Of course any number multiplied by Zero is equal Zero in the natural numbers. The idea then is to define the function in a way, that $m(a,b)$ via projection and $a$ are added resulting in the product of $a$ $b$ increasing by the magnitude of $a$.

3.4. **Factorial function.** Another nice example for using the concept of primitive recursive functions for building a well-known operation is the factorial function, let's call it $F$. which is defined as $F : \mathbb{N} \to \mathbb{N}$.

(1) Define $F(0) = g() = s(()) = 1$

(2) Apply primitive recursion and the projection functions:

$$F(x+1) = h(x, F(x)) = m(I_2^2(x, F(X)), s(I_1^2(x, F(X))))$$

(3) $m(I_2^2(x, F(X)), s(I_1^2(x, F(X)))) = F(x) \times (x+1)$

Using the multiplication function, the only thing that needs to be done to obtain the factorial function is to multiply the factorial of $x$ with it's successor. This can again be achieved via the projection function, taking the second component of the pair $(x, dF(X))$ and multiplying it with the successor of $x$ (applying the the successor function on the first component of the pair given above).

It is quite interesting to see how much more complicated easy algorithms seem, if one writes them down in the language of primitive recursive functions, compared to the algorithm kids learn in great school to calculate the value of these functions.

Now that we have taken a closer look at primitive recursive functions it is time to add the $\mu$-operator in order to obtain the smallest class of functions, that contain all calculable functions. One could ask why it is necessary to expand the comfortable (because total) class of primitive recursive functions. It indeed would have been convenient result, had it turned out that all calculable functions could be built up this way. As mentioned above

David Hilbert even assumed in 1926, see [8] that every calculable function can be pored into the shape of a primitive recursive function. But Ackermann functions for example grow faster, than any primitive recursive function (so even faster than any exponential functions), which will be discussed right after introducing the $\mu$-operator.

Adding the $\mu$-operator, while expanding the array of functions, that can be investigated, will shift the functions from being total (all primitive recursive functions are total functions) to dealing with partial functions. Worse then that: even if the domain of a partial function is well-defined the function will not always return a value, as an example for such an incalculable number the Halting number respectively the Halting problem will be discussed and a proof of the later will be given after introducing the universal function later.

### 3.5. **Search operator.**

**Definition 5.** The search operator $\mu$ for a $(k+1)$-place partial function $f : \mathbb{N}^{k+1} \to \mathbb{N}$ is defined as

$$\mu b[f(\vec{a}, b) = 0] = \begin{cases} \text{the least b, so that } f(\vec{a}, b) & \text{if } f(\vec{a}, b) = 0 \text{ defined for all } z < b \\ \uparrow & \text{otherwise} \end{cases}$$

where $\uparrow$ means, that the function is not defined.

So what the $\mu$-operator does, is search for the smallest zero of a function considering its first k-places and keeping free its last. If that functions does not have a smallest zero or there is an undefined value before reaching the smallest zero, than the $\mu$-operator is not defined, and the calculation will not terminate. Let us look at a simple example.

We have a function

$$f : \mathbb{N}^2 \to \mathbb{N} \text{ and } f(x, y) = \begin{cases} x + y & \text{if } x + y > 3 \\ 0 & \text{if } x + y \leq 3 \end{cases}$$

Then applying the $\mu$-operator will give us a functions $\mu[f(x,y)]$ with

$$f : \mathbb{N} \to \mathbb{N}.$$

Let us illustrate this in a grid.

| $x^y$ | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| 0 | 3 | 2 | 1 | 0 | 0 | 0 |
| 1 | 2 | 1 | 0 | 0 | 0 | 0 |
| 2 | 1 | 0 | 0 | 0 | 0 | 0 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 0 | 0 | 0 |

$\mu f(1) = 3$

$\mu f(2) = 1$

$\mu f(3) = 0$

$\mu f(4) = 0$

$\vdots$

Here the $\mu$-operator gives us a value for all of the domain. But it is very simple to come up with an example for a function, where this is not the case. For example let a(x,y)=x+y with $a : \mathbb{N} \to \mathbb{N}$. In this case $\mu a(1)$ is not defined, because the operator will not find a zero for $x = 1$. But with unbounded search, the program will not terminate. This is one problem that occurs. Obtaining the class of all calculable functions comes with a price: the class of general recursive functions is not total, meaning that not every general recursive function is total, e.g. the minimisation of the successor function. And some well defined algorithms do not halt (a case that can't occur in the class of primitive recursive functions).

3.6. **Ackermann function.** As mentioned above not every calculable function has the shape of a primitive recursive function. With the $\mu$-operator at hand it is possible to construct functions like the Ackermann function, that expand the class of primitive recursive functions to the class of general recursive partial functions. The following definition was

developed by Peter and Robinson and is a two-argument function(opposed to Ackermann's original version with three arguments), see [5].

**Definition 6.** Let $m, n$ be integers. Then the Ackermann function is defined by recursion through:

- $A(0, n) = n + 1$
- $A(m + 1, 0) = A(m, 1)$
- $A(m + 1, n + 1) = A(m, A(m + 1, n))$

Now that we defined the Ackermann functions, it seems interesting to see just how fast it grows and look at a few big values. Since the Ackermann function, defined like above, is a two-place function, we can use a chart for the first values.

| n/m | 0 | 1 | 2 | 3 | 4 | ... |
|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | ... |
| 1 | 2 | 3 | 4 | 5 | 6 | ... |
| 2 | 3 | 5 | 7 | 9 | 11 | ... |
| 3 | 5 | 13 | 29 | 61 | 125 | ... |
| 4 | 65533 | $2^{65536} - 3$ | ... | ... | ... | |

The first row is rather easy to explain. By definition the value of the Ackermann function is just $n + 1$ whenever $m = 0$. Also the first column only contains the same value as the cell on it's upper right corner, since the value for $n = 0$ and $m > 0$ is defined that way. Going row by row we always calculate the actual value of looking at a value in the line above. So for example if $A(3, 2)$ is the value in question, we first calculate $A(3, 1)$ in order to obtain $A(2, A(3, 1))$ which is listed above. The growth of each row, can be described as the following:

(1) For a fixed $m = 0$ there is a linear growth of the values as $n$ increases. For every increase in $n$ by one, the value also increases by one.

(2) For a fixed $m = 1$ the second row, the value of the function behave the same way.
(3) For a fixed $m = 2$ In the third row, the pace of growth is still linear, but it increases by factor 2 (plus 3 as an additive constant).
(4) For a fixed $m = 3$ the Ackermann function already grows exponentially with respect to $n$.
(5) For a fixed $m = 4$ and it is not possible to express the growth rate expressible with common operations. Using hyper operators or Knuth's upper arrows can help. Trying to express these values with common decimal representation would be pointless.

Using this chart and some imagination, picturing a growth rate faster than any exponential function is possible. This put an end to Hilbert's hypothesis. Ackermann's function like every primitive recursive function is a total function, but it is not primitive recursive, since it requires the searchoperator, which comes with the cost of uncalculability in other cases.

## 4. Register Machines

Another way of putting the informal idea of effective calculability on solid and rigorous grounds is the concept of register machine. As announced earlier and demonstrated in the following section, although it might not seem similar to the concept of general recursive functions, both are equivalent, meaning they define the exactly same class of partial functions. First of all we will define what a register machine and a program for it are. The following definitions and theorems are closely inspired by [1] and [6].

**Definition 7.** A program for a register machine with $0, ..., K$ is a finite list of instructions $\langle I_0, ..., I_n \rangle$ chosen from the ones below:

(1) $I_r$, where $0 \leq K$: increase r by 1 and move to the next instruction;
(2) $D_r$, where $0 \leq K$:
    if the content of $r$ is $> 0$ the decrease its content by 1 and skip one instruction;

if the content of the register $r$ is 0, the do nothing and go to the next instruction;

(3) $J_q \in \mathbb{Z}$; All registers are left unchanged, and:
if $q > 0$ the jump to the $qth$ instruction following the current one;
if $q = 0$ then enter a loop

if $q < 0$ the jump to the $|q| - th$ instruction preceding the current one.

So there are only these three very simple instructions and a starting condition, containing the information of the content of every register, all holding a finite not negative number. The increment instructions increases the natural number by the magnitude 1 in the addressed register and goes on to the next instruction in the program. The decrement instruction decreases the number in the addressed register by the magnitude 1 and than skips one instruction and executes the next instruction, if the addressed register is not already 0. In later case it does not decrease the number in the register, but just proceeds with the next instruction, leaving the content of every register unchanged. And finally Jump $n$ moves $n$ steps in the list of instructions, depending on the sign either back or forward. All the register remain the same and the instruction jumped to is executed. he machines comes to a good halt, if it tries to reach instruction $k+1$ of $k$ instructions, which of course does not exist, revealing the value of the partial function in register 0, which is initially always equal to 0. Bad halts appear if the program tries to reach another non existing instruction.

Now we will define the part above concerning partial function, that are computable by a register machine, more formally.

**Definition 8.** A partial function $f : \mathbb{N}^k \to \mathbb{N}$ is RM-computable, if there is a RM and a RM-program $\langle I_0, ...; I_n \rangle$ such that whenever $\vec{x} = (x_1, ..., x_k) \in \mathbb{N}^k$ and he machine starts with initial configuration: $x_1, ..., x_k$ in registers $1, ..., k$ respectively, and 0 in all other registers, then:

(1) if $\vec{x} \in dom(f)$, then the computation terminates with values $f(x_1, ..., x_k)$ in register 0, while the program is looking for instruction $n + 1$
(2) if $\vec{x} \notin dom(f)$, then the computation never halts.

Before showing the equivalence of RM-computability and general recursive functions, let us look at a few example of how a program for easy operations looks like. First we want to find a program that can add two natural numbers $a$ and $b$. Let us start with register 1 containing $a$ and register 2 containing $b$. We want to receive the result of $a + b$ in register 2, just by adding the content of register 1 to register 2. Here is such a list of instructions:

(1) $\langle D1\rangle$
(2) $\langle J3\rangle$
(3) $\langle I2\rangle$
(4) $\langle J - 3\rangle$

The first instructions decreases the natural number $a$ by one and the machine goes on to instruction 3. It increases the number $b$ by one and goes onto instruction 4, which makes it jump back to instruction one. This happens as long as the register one is not 0. As soon as register one is empty the machines proceeds to instruction 2, which results in a halt, because there is no fifth instruction. This is just one very simple way of finding a program that can add two natural numbers, although it is strictly speaking not a register machine program, since this would require the value of the addition showing up in register 0 after the computation terminates.

Another example for a RM-computable functions is the Successor function. The one-place successor function $S(n) = n + 1$ for $f : \mathbb{N} \to \mathbb{N}$ can be computed by the following program,where we start out with the number to be increased in register 1:

(1) $\langle D1\rangle$
(2) $\langle J3\rangle$
(3) $\langle I0\rangle$
(4) $\langle J - 3\rangle$

(5) $\langle I0 \rangle$

The registers 0 is always empty at the start of the computation. The program moves the content of register 1 step by step to register 0 using the jump instruction. Once register 1 is empty the program will move on to the second instruction, then jumping to the fifth, and halting, since there is no sixth instruction.

## 5. $GR \subseteq RM$

The proofs and definitions in this section closely follow [1]. The first step to show the equivalence of the class of RM-computable functions and the class of general recursive function will be showing that every general recursive function is indeed RM-computable. To do that it will be necessary to show, that the class of RM-computable functions is closed under composition, primitive recursion and finally also closed when adding the $\mu$-operator.

**Definition 9.** Let $x$ be a $k$-place partial function, $R$ is a program and $z_1, ...., z_k$ are distinct natural numbers and $s, t \in \mathbb{N}$. $R$ computes $g$ from $z_1, ...., z_k$ $x$ if whenever we start a register machine with $R$ and with $y_1, ...., y_k$ in registers $z_1, ...., z_k$, then, never mind what is in the other registers initially, we have the following result:

(1) If $x(a_1, ...., a_k)$ is defined, then the computation comes to a good halt, with $x(a_1, ...., a_k)$ in register $t$. The first register $0, 1, ..., s-1$ will have the same content as in the starting configuration, expect potentially for register $t$.
(2) If $x(a_1, ...., a_k)$ is undefined, the the computation never halts.

A proof, that one always can find a programm $R$ running the computation defined above, can be found in [1].

**Theorem 10.** *The class of RM-computable partial function is closed under composition. Whenever we have partial functions $r, s_1, s_2, ..., s_n$ which compose the partial function $t$, and we have RM-programs that compute $r, s_1, s_2, ..., s_n$, we can create a RM-program that computes $t$.*

*Proof.* Suppose that the k-place function $t : \mathbb{N}^k \to \mathbb{N}$ is created by the composition from $r, s_1, s_2, ..., s_n$: $t(\vec{x}) = r(s_1(\vec{x}, ..., s_n(\vec{x})$

(1) Calculate $s_1$ from $1, ..., k$ to $k+1$, preserving $k+1$

(2) Calculate $s_2$ from $1, ..., k$ to $k+2$ , preserving $k+2$

(3) ...

(4) Calculate $s_n$ from $1..., k$ to $k+n$, preserving $k+2$

(5) Calculate $r$ from $k+1, ..., k+n$ to 0, preserving 0.

For the program to halt, we need $s_1(\vec{x}), ..., g_n(\vec{x})$ to be defined and $f$ to be defined at $\langle s_1(\vec{x}), ..., g_n(\vec{x})\rangle$ □

Whenever we have a composition $t$ of partial functions we start out calculating every inner values $s_1, s_2, ..., s_n$ which are preserved in register $k+1, k+2, ..k+n$, which initially are zero. These values are then used as starting register for the outer value $r$, which after terminating preserves the value of $t$ in register 0.

The next step to show, that every general recursive function is RM-computable, is to prove, that the class of RM-computable partial functions is closed under primitive recursion.

**Theorem 11.** *The class of RM-computable function is closed under primitive recursion.*

*Proof.* Let h be the partial (n+1)-place function, defined with the following operations:

(1) $h(\vec{x}, 0) = f(\vec{x})$

(2) $h(\vec{x}, y+1) = g(h(\vec{x}, y), \vec{x}, y)$

If we assume that $g$ and $f$ are RM-computable functions, then we start out with the following register content:

- register (0) $h(\vec{x,t})$
- register (1) $x_1$

$\vdots$

- register $(n)$ $x_n$
- register $(n+1)$ $y-t$ (Initial value $y$)
- register $(n+2)$ $t$ (Initial value 0)
- register $(n+3)$ $y-t$ work space

$\vdots$

The RM-program executed is the following:

Calculate $f$ from $1, ..., n$ to 0, preserving $n+3$

(1) $\langle D, n+1\rangle$ Begin loop.

(2) $\langle J, \square+3\rangle$ Halt when done.

Calculate g from $0, 1, ..., n, n+2$ to 0, preserving $n+3$

(3) $\langle I, n+2\rangle$ $t := t+1$

(4) $\langle J, -(\square+3)\rangle$ Repeat loop.

(5) HALT

□

The above demonstrated proofs assure us, that every primitive recursive function is RM-computable, since we have shown that all tools to built a primitive recursive function, are RM-computable. Like in many other cases, a loop using decrement and jump is being constructed. As soon as the register that is being decremented becomes empty, the program will go in a new mode: executing the next instruction instead of the one after that, eventually coming to halt. What remains to be shown is, that once the $\mu$-operator is added, the partial functions will also remain RM-computable.

**Theorem 12.** *Every general recursive function is a RM-computable function.*

*Proof.* Consider a function $h(\vec{x}) = \mu y[g(\vec{x}, y)$ and suppose $g$ is a RM-computable. Now we use the following program:

(1) $\langle D, n+1\rangle$

(2) $\langle J, 3\rangle$

(3) $\langle I, 0\rangle$

(4) $\langle J, -(\square + 3)\rangle$

The program uses following registers:

- register (0) $y$
- register (1) $(x_1)$

$\vdots$

- register $(n)$ $x_n$
- register $(n+1)$ $g(\vec{x}, y)$
- register $(n+2)$ and the following registers are work space.

□

This program, like search with the $\mu$-operator, may never halt. Since $y$ is kept in register 0 only if the function $g$ has a smallest Zero, the program will eventually skip to the second instruction, causing it to seek the first nonexisting instruction and coming to a good halt.

## 6. $RM \subseteq GR$

6.1. **Encoding Register Machine Programs.** The proofs and definitions in this section again closely follow [1] and [6]. What was shown until now is, that $GR \subseteq RM$, where $GR$ is the class containing all general recursive functions and $RM$ is the class containing all RM-computable functions. To show that the classes are equivalent, it will be necessary to

show $RM \subseteq GR$, meaning that every RM-computable function is general recursive. As it turned out, this is a little bit more difficult. One can use the concept of the universal program respectively the universal function and other special functions to achieve this.

$\Phi(e, x) =$ the result of applying the program coded by e to input x

The basic idea is the following: every set of instructions, in our case a set of RM-instructions can be encoded into a single number $e \in \mathbb{N}$ (how this can be achieved will be discussed in detail below) First it will be necessary to define the bracket notation, used to map the Gödel number.

**Definition 13.** The "bracket-notation" is defined as follows: Let $b : \mathbb{N}^k \to \mathbb{N}$ be a partial function. Then

(1) $[\ ] = 1$

(2) $[x_0, x_1, ..., x_k] = 2^{x_0+1}3^{x_1+1}...p_k^{x_k+1}$

After decoding the instructions, they are applied to input $x \in \mathbb{N}$, which carries all the information about the content of the registers before the computation starts. In order to do this the first step is to assign a Gödel number to all for types of instructions a RM uses:

**Definition 14.** Encoding Instructions

We define the Gödel number for each instruction

(1) $\#(Ir) := [0, r]$, where $Ir$ is increment instruction.

(2) $\#(Dr) := [1, r]$, where $Dr$ is the decrement function.

(3) $\#(Jq) := [2, q]$ if $q \geq 0$ and

(4) $\#(Jr) := [3, |q|]$ if $q < 0$.

So for example if we know that 18 is the Gödel number of an instruction, we can look at the prime factorization of $18 = 2 \cdot 3 \cdot 3$. Then we look at $[x, y] = 2^{x+1} \cdot 3^{x+1}$ and we find out that the instruction is $[0, 1]$, which means increment register 1 by 1.

Also if $a$ is the Gödel number of a RM-instruction and we look at its prime factorization, then the exponent of 2 tells us what kind of instruction it is, and the power of 3 tells us the address of the register or the size of the jump. Of course some numbers can't be interpreted as an instruction of a RM-program, for example 10, because, there is no instruction, which contains primes greater than 3.

The next step is to give every program, which is a finite list of instructions, a Gödel number. If we have a list of instructions $P = (r_1, ..., r_n)$ then $[\#r_1, ..., \#r_n] = e$ gives us the Gödel number of the program.

**Definition 15.** Encoding a program

Let $P$ be a list of RM-instructions, then

$$\#(P) := [\#(I_0), ..., \#(I_n)]$$

In particular if $P$ is empty, then $\#(P) = 1$.

Let us again look at an example given a program from the previous chapter, namely the Successorfunction:

(1) $\langle D1 \rangle$
(2) $\langle J3 \rangle$
(3) $\langle I0 \rangle$
(4) $\langle J - 3 \rangle$
(5) $\langle I0 \rangle$

Now we want to find the Gödel number of every instruction.

$$[\#D1, \#J3\#I0\#J-3, \#I0] = [[1,1],[2,3],[0,0],[3,3],[0,0]] = [36, 648, 6, 1296, 6] =$$

$$2^{37} \cdot 3^{647} \cdot 5^6 \cdot 7^{1296} \cdot 11^6 = e^{1861499664 \cdot ln(17)} = e^{5274025688}$$

This number is rather big for a five line program, but since we are still in a fully theoretical setting, where computational complexity does not matter, it is perfectly fine. Also notice that the number is unique, in the sense that every possible program is assigned a number bijectivley. The first instructions is encoded in the exponent of 2, the second instruction in the exponent of 3 and so on.It is important to understand, that up to this point only operations which are primitive recursive were used. In trying to show that every RM-computable function is general recursive, it will remain crucial that all operations that are being used, are primitive recursive, despite the $\mu$-operator, which expands the class of primitive recursive functions to that of general recursive function.

(1) Let $\langle x_0, ..., x_n \rangle$ be a sequence in $\mathbb{N}$. Then $\langle x_0, .., x_n \rangle \to [x_0, ..., x_n] := \prod_{i=0}^{n} p_1^{x_i+1}$ is primitive recursive.

(2) For $e \in \mathbb{N}, (e)_{k^*}$ is defined as the power of $p_k$ in prime factorization of $p$; The function is also primitive recursive.

(3) Thus the function $(e)_k = (e)_{k^*} - 1$ is also primitive recursive.

(4) Note that if $e = [x_0, ..., x_n]$, then $(e)_k = x_k$ for $0 \leq k \leq n$.

6.1.1. *Memory Number.* The next step is to encode the content of all registers into one single number. It is important to mention, that although there is a infinite number of registers, at any given moment of the computation only a finite amount of these registers are nonzero. All the registers containing zero do not factor in, as the definition below shows.

**Definition 16.** Memory number

If for each $i$, let $x_i$ be the content of register $i$, the $m = \prod_{i=0}^{K} p_i^{x_i}$ encodes the content of all registers.

Again, given any natural number $n$: if we know that $n$ is the memory number of a RM, then by applying prime factorization we can find out the content of every register fairly quickly. For example take the number $125 = 5^3$. Now we now that all the register but register 5 are empty and register 5 contains the number 3. Or for example, if we know that the memory number is $24 = 2^3 \cdot 3^1$ we know that register 0 holds 3 register 1 holds 1 and all other register are empty.

Now of course the content of the register may alter after every step the RM makes, e.g. processing a decrement instruction. In order to get keep track of this, it will be necessary to introduce a new function, namely the memory function, that combines the current state of the register (given by the memory number) with the Gödel number of the instruction that is going to be carried out by the RM, returning a value equal to the memory number of the new state the registers are in after executing the instruction.

**Definition 17.** Memory Function

$$mem(m, c) = \begin{cases} mp_{(c)1}, & \text{if } (c)_0 = 0, c \neq 0 \\ m/p_{(c)_1}, & \text{if } (c)_0 = 1, \text{ and } p_{(c)_1} | m \\ m & \text{if otherwise} \end{cases}$$

Thus, if $m$ is a memory number and $c$ is the Gödel code of an instruction, then $mem(m, c)$ computes the memory number of the RM after the execution of $c$.

It is easy to understand that the memory number does not change, if $c$ is a jump-instruction, since these leave the content of every register unchanged. If $c$ is an increment instruction, all that has to be changed is the exponent of the prime factor assigned to the

register. Since the second component $(c)_1$ of an instruction tells us the addressed register, all we have to do is multiply $m$ with that prime factor. Similarly if $c$ is a decrement instruction it will be necessary to divide $m$ by exactly that prime factor. One could ask, what if that register is empty and that prime factor therefore is not a divider of m. Well in case the register is empty, the decrement instructions does not change the content of the registers (by its own definition) and the program just moves on to the next instruction.

Here is a example for an arbitrary memory number $d$ and $c = 144 : mem(d, 144) = m$ because the instruction encoded by the number $144 = 2^4 \cdot 3^2$ and following $[x, y] = 2^{x+1} \cdot 3^{y+1}$ we receive the instruction $\langle J, -1 \rangle$, so all the registers and the memory number remain the same.

6.2. **Location counter.** The next step is to define, what is commonly called a location counter. The location counter keeps track of which instructions is currently being executed. Again this information should be contained in a single number.

Given the number $k$ of an instruction of a RM-program, a memory number $m$ and the code $e$ of the RM-program, the function, which is defined below, $loc(k, m, e)$ computes the number of the instruction, which is to be executed after $I_k$, where the Gödel code of $I_k$ is $(e)_k$.

**Definition 18.** Location Function

$$loc(k,m,e) = \begin{cases} k, & \text{if } k \geq lh(e) \text{(already stopped)} \\ k+2, & \text{if } if((e)_k)_0 = 1, p_{((e)_k)_1} | m, k+2 \leq lg(e) \\ min(k + ((e)_k)_1, lh(e), & \text{if}((e)_k)_0 = 2 \text{ (jump forward)} \\ k - ((e)_k)_1, & \text{if}((e)_k)_0 = 3, ((e)_k)_1 \leq k \text{ (jump back)} \\ lh(e), & \text{if}((e)_k)_0 = 3, ((e)_k)_1 > k \text{ (bad jump)} \\ k+1, & \text{if otherwise} \end{cases}$$

with lh$e = m + 1$, for a program with $m$ instructions.

If the program has $m$ instructions, than as soon as the location counter equal $lhe$, the program comes to a good halt, reaching the first non existing instruction. For an example let us look at the value $t$ of the location number of the following argument. $loc(0, m, 32) = t$. The prime factorization of $32 = 2^5$. So our program has only one instruction

$$[4] = \langle D, 0 \rangle$$

Since the register 0 is always 0 at the beginning of the computation, the program tries to reach the next instruction, which does not exist and comes to a good halt, returning the value 0. The function defined above, will return the value 0, because the program only has one instruction. All the location function does, is look at the program encoded in $e$ and figure out what kind of instructions is being executed. The memory number only is necessary to determine whether a decrement instruction leads skipping or not skipping an instruction depending on if the content of the register is 0 or not. Now we have all the tools to construct the universal program, a effectively calculable two-place partial function, that given any RM- program and a starting configuration of the registers will assign a natural number to the combination of program and input.

## 6.3. **Universal Program.**

**Definition 19.** A universal program is a program which computes a two-place partial function, which produces the result of applying the program encoded by e to input $x$. $lhe = m + 1$ Use of register:

- [0] := number of remaining instruction
- [1] := e, code of the program
- [2] := x, input value
- [3] := location counter of the program
- [4] := memory
- [5] := $(e)_{[3]}$ The Gödel number of the next instruction

All the other registers are empty at the beginning of the program. Now this is the starting configuration of the universal program. Next we need a list of instructions the program executes:

(1) Calculate $3^x$ from 2 to 4 preserving 4 (initialize memory)

(2) Calculate $lh(e) - [3]$ from $1, 3$ to 0 preserving 5 (start loop here)

(3) $\langle D, 0\rangle$

(4) Jump to register 9 (Exit loop)

(5) Calculate $(e)_{[3]}$ from $1, 3$ to 5 preserving 5 (get the next instruction)

(6) Calculate $loc$ from $3, 4, 1$ to 3 preserving 6 (update location)

(7) Calculate $mem$ from $4, 5, 6$ preserving 6 (update memory)

(8) Jump back to register 2 (start loop again)

(9) Calculate $(m)_{0^*}$ from 4 to 0 preserving 6

Now there are three different scenarios, that can occur after letting the program $e$ calculate with input $x$:

(1) Let $e = \#(P)$ and $P$ computes a partial function $f$. If $x \in dom(f)$ the the universal program halts with result $f(x)$. If $x \notin dom(f)$ then the universal program does not stop.

(2) If $e$ is a code of a program, which does not compute a partial function (e.g. $P$ has a bad jump), then the universal program will set virtual location counter to $lh(e)$ and will stop (good halt) with output $(m)_{0^*}$ (where $m$ is the memory number at the time)

(3) If $e$ is not the Gödel number of a program, the universal program will start running nevertheless. It will either stop, or not, depending on $e$.

The halting problem says, that given a program $e$ and an input $x$ it is impossible to decide, if the program comes to a halt, even if it is well defined. More concerning this will be discussed shortly. Considering this the universal program computes a two-place function

$$\Phi(e,x) = \begin{cases} \text{output of } \Phi(e,x), & \text{if } \Phi(e,x) \text{ halts} \\ \uparrow & \text{if otherwise} \end{cases}$$

So for each $e \in \mathbb{N}, \Phi(e,x)$ is a RM-computable partial function, which will further on be denoted as $[[e]]$. If $e = \#(P)$ and $P$ computes a partial function $f$, the $[[e]](x) = f(x)$ for all $x$. The number $e$ is called the index of $f$.

**Theorem 20.** *Enumeration Theorem*

*(1)* $\Phi$ *is a RM-computable two-place function.*

*(2) For each* $e$, $[[e]]$ *is a RM-computable one-place function.*

*(3) Every one-place RM-computable function if* $[[e]]$ *for some* $e$.

Thus $[[0]], [[1]], ...$ is a list (with repetitions) of all unary RM-computable one-place partial functions. This concept can easily be generalized for n-place functions see [1] or [7].

6.3.1. *Snapshot Function.* To show that every RM-computable partial function is general recursive another tool is necessary. Given $e, x$ a parameter of time $t$ can be introduced. Then the Gödel number, taken from the combination location counter memory number uniquely identifies the status of the computation. The goal is to define the snapshot function, using primitive recursion.

**Definition 21.** Snapshot
Let $e$ be the encoding of a RM-program and $x$ the starting configuration oft a RM, $k$ the location counter and $m$ the memory number of that program. Then,

$$snap(e, x, t) = [k, m]$$

So $snap(e, x, 0)$ gives us the initial condition of the universal program, with $k = 0$ and $m = 3^x$ resulting in the value $[0, 3^x]$. This allows us to to define the snapshot function using primitive recursion:

(1) $snap(e, x, 0) = [0, 3^x]$

(2) $snap(e, x, t + 1) = [loc(k, m, e), mem(m, (e)_k)] =$

$$[loc((snap(e, x, t))_0, (snap(e, x, t))_1, e), mem((snap(e, x, t))_1, (e)_{(snap(e,x,t))_0})].$$

As stated above at step 0 the location counter is 0 and the memory number is $3^x$. The next instruction to execute is $(e)_k$ provided $k < lh(e)$. Then the memory number changes from $m$ to $mem(m, (e)_k$. The location counter changes from $k$ to $loc(k, m, e)$. Since the snapshot function can be defined by primitive recursion and only consists of primitive recursive function, it is a primitive recursive function itself. The snapshot function is also a total function, even if $e$ is not an encoded list of RM-instructions.

6.3.2. *Running time function.* So the snapshot function gives us a unique number that holds information of the location counter and memory depending on time respectively the state of the computation. If the location ever says that the computation terminated, then the exponent of the prime factorization of the snapshot value, will be greater or equal to the number of the instructions in a program. We can define a running time relation $T$:

**Definition 22.** Running Time Relation

$$T = \{\langle e, x, t\rangle | (snap(e, x, t))_0 \geq lh(e)\}$$

Now by applying the $\mu$-operator we can measure the running of the computation:

- $\langle e, x\rangle \to \mu t T(e, x, t)$
- $\langle e, x\rangle \to \mu t([[e]](x) \downarrow in \leq t$ steps

Again the search operator may run forever here and we can not decide, whether this will happen or not. This problem is also known as the halting problem. But the good news is, that we definitely have a general recursive partial function, because all that was done is adding the $\mu$-operator, all the other functions are primitive recursive.

**Theorem 23.** *Normal Form Theorem*

*For any $e$ and $x$,(where $e$ represents a list of RM-instructions and $x$ is the starting configuration of a RM)*

$$[[e]](x) = ((snap^n(e, x, \mu t T(e, x, t)))_1)_0^*$$

*(1)* $(snap^n(e, x, \mu t T(e, x, t)))_1)$ *decodes the memory number.*

*(2)* $((snap^n(e, x, \mu t T(e, x, t)))_1)_0^*$ *extracts the output in register 0, the value of the function.*

Every function used on the right side of the equations is primitive recursive, but the $\mu$-operator. But adding the $\mu$-operator just expands the class of the primitive recursive

functions to the class of general recursive partial functions. Putting this theorem together with the proof that every general recursive partial function is RM-computable, shows both classes are equivalent.

6.4. **Halting problem.** Now a natural question given the code of a program $e$ and a starting configuration $x$ is if this program will halt and return a result. The following definition, theorem and proof will closely follow [6] Given a partial function $f(x)$ an input $x$ and a code $e$, $s$ is a number encoding the computation of the program $e$ on input $x$, if $f(x)$ is defined. Now there is a relation $T(e, x, s)$ and a primitive recursive function $U(s)$, so that

$$f(x) \simeq U(\mu s T(e, x, s))$$

So only a single unbounded search is necessary to determine the smallest $s$, so that the relation holds.

**Definition 24.** If $f$ is a partial recursive function, any $e$ for which the equation in the normal form theorem (given above) holds, is an index of $f$. The number $e$ can be used as a index for the partial functions, an we can write $\varphi_e(x)$ instead of $f(x)$. Given a number $e$, the normal form theorem states that

$$\varphi_e(x) \simeq U(\mu s T(e, x, s))$$

is partial recursive.

**Definition 25.** Halting function The halting function is defined by

$$h(e, x) = \begin{cases} 1 & \text{if } \varphi_e(x) \downarrow \\ 0 & \text{otherwise} \end{cases}$$

Here both cases are included: the one where $\varphi_e(x)$ is undefined and the case where $e$ just is not the index of a partial function. (The later case will not be mentioned separately in the following proof.)

**Theorem 26.** *The halting function $h$ is not partial recursive.*

*Proof.* If $h$ were partial recursive, we could define

$$d(y) = \begin{cases} 1 & \text{if } h(y,y) = 0 \downarrow \\ \mu x(x \neq x) & \text{otherwise} \end{cases}$$

Since no number $x$ satisfies $x \neq x$, there is no $\mu x$ with this condition, and so $d(y) \uparrow$ iff $h(y,y) \neq 0$. This implies that :

(1) $d(y)$ is defined iff $\varphi_y(y)$ is undefined.

(2) $d(y)$ is undefined iff $\varphi_y(y)$ is defined.

If $h$ was partial recursive, that would mean that $d$ is too. Now taking in consideration the normal form theorem it follows, hat there is a index $e_d$. The next step is to look at the value of $h(e_d, e_d)$.

(1) If $h(e_d, e_d) = 1$ then $\varphi_{e_d}(e_d)$ is defined. But $\varphi_{e_d} \simeq d$, and $d(e_d)$ is defined iff $h(e_d, e_d) = 0$. So $h(e_d, e_d) \neq 1$.

(2) If $h(e_d, e_d) = 0$ then $\varphi_{e_d}$ is undefined. But again $\varphi_{e_d} \simeq d$, and $d(e_d)$ is undefined iff $\varphi_{e_d}(e_d)$ is undefined.

This is a contradiction to the assumption that $h$ is partial recursive. □

Since all the calculable partial functions are general recursive, the halting functions is an example for a not calculable partial function. There is a certain kind of trade-off that takes place here: by leaving the class of primitive recursive functions, which are all total and calculable, to obtain a class that holds every calculable function, uncalculability is obtained. This circumstance will be discussed in the conclusion of the thesis.

After showing the equivalence of two of the most famous formalizations of effective calculable functions and the fundamental theoretical limit, namely the halting problem, we will turn to another way of formalizing the concept of effective calculability. A proof for this can be found here. Turing machines are equivalent to the two discussed above concepts. This thesis will not contain a proof of latter claim, since it would go beyond its scope, but a proof can be found here [2]. Turing machines will be used to demonstrate another aspect of computability, namely practical boundaries, such as time and space complexity.

It can be shown that Gödel's incompleteness theorem and Turing Halting problem are equivalent. See [10]

## 7. Turing machines

To begin it will be necessary to define exactly what a Turing machine is, and how it operates. The following definitions and proofs are closely oriented on [3] and [11].

The thesis will present the definition of a $k$-tape Turing machine and go on to show that every $k$-tape is equivalent to a particular one tape Turing machine.

**Definition 27.** A Turing machine (TM) with k work tapes is a pair $(S, \delta)$ where $S \neq \emptyset$ is a finite set, referred to a a set of states, such that $S_{start}$ (the starting state) and $S_{halt}$ (a stopping state) are members of $S$ and

$$\delta : S \times \{\triangle, \square, 0, 1\}^k \to S \times \{\triangle, \square, 0, 1\}^k \times \{1, 0, -1\}^k$$

is the transition function, which has the following properties:

Let $\delta(s, \overline{a}) = (s', \overline{b}, \overline{m})$ where $\overline{a} = a_1, ..., a_k$ (the current content of the tape), $\overline{b} = b_1, ..., b_k$ are in $\{\triangle, \square, 0, 1\}^k$ and $\overline{m} = m_1, ..., m_k$ is in $\{1, 0, -1\}^k$.

Then for each $i \in [k]$ we have :

(1) $a_i = \triangle$, iff $b_i = \triangle$

(2) if $a_i = \triangle$ then $m_1 \neq -1$

(3) if $S = S_{halt}$ then $s = s', \overline{a} = \overline{b}$ and $m_i = 0$

The work tape is an infinite array of cells, each containing $0, 1, \square$ or $\triangle$. The machine furthermore has a head (or k-heads if it is a k-tape TM) scanning one cell at a time. At the start the TM is always in its initial state head starts to scan the cell number 0. The input $x = x_1...x_n \in \{0, 1\}$ is written bit by bit in cells $1, ..., n$ respectively cell 0 contains the symbol $\triangle$:

(1) $\triangle$ never changes and does not appear anywhere else.

(2) If a head scans $\triangle$, it does not move to the left.

Also if $\delta(s, \overline{a}) = (s', \overline{b}, m)$ this means that the machine is in state $s$ and changes $\overline{a}$ with $\overline{b}$ and then moves according to $m \in \{0, 1, -1\}$ and passes to state $s'$. Last, if $s_{halt}$ is reached, the computation stops.

So these are basically all the essential rules a Turing machine is characterised by. The next step is to give a definition of what a configuration graph is.

**Definition 28.** Let $k, n \in \mathbb{N}, k > 0$. Consider a $k$-tape TM $\mathbb{M} = (S, \delta)$ and an input $\overline{x} = x_1 x_2 ... x_n \in \{0, 1\}$.

(1) A tuple $(s, \overline{j}, \overline{c}) \in S \times \mathbb{N}^k \times (\{0, 1, \square, \triangle\}^k$ is called a configuration of $\mathbb{M}$.

(2) We say that $(s, \overline{j}, \overline{c})$ is halting if $s = s_{halt}$.

Here $\overline{j} = j_1 ... j_k \in \mathbb{N}^k$ is to be interpreted as the current location of the heads and $\overline{c} = c_1 ... c_k$, where $c_i \in \{0, 1, \square, \triangle\}^{\mathbb{N}}$ is a countably infinite array of the symbols $\{0, 1, \square, \triangle\}$, that describes the content of the $i - th$ tape. Thus a configuration gives a complete information about the Turing machine in a given state.

**Definition 29.** Starting Configuration

Let $0^k$ denote the $k - tuple$ 0...0. The starting configuration of $\mathbb{M}$ on $\overline{x} = x_1 ... x_n$ is $(S_{Start}, 0^k, \overline{c})$, where $\overline{c} = c_1 ... c_k$ is defined as follows.

For all $i \in [k]$ (referring to the row index), $j \in \mathbb{N}$ (referring to the column index)

$$c_i(j) = \begin{cases} \triangle & \text{if } j = 0 \\ \square & \text{if } j > 0, i > 1, or j > n, i = 1 \\ x_j & \text{if} i = \triangle, j = 1, ..., n \end{cases}$$

**Definition 30.** Successor Configuration
A configuration $(s', \overline{j}', \overline{c}')$ is a successor configuration of $(s, \overline{j}, \overline{c})$, if $\exists \overline{m} = (m_1, ..., m_k) \in \{0, 1, -1\}^k$ such that $\forall i = 1, ..., k$ we have:

(1) $\delta(s, c_1(j_1)....c_k(j_k)) = (s', c'_1(j1)...c'_k(j_k), \overline{m)}$

(2) $j'_1 = j'_1 + m_i$

(3) $c_i(l) = c'_i(l) \, \forall l \neq j_i$

We are interested in the collection of all configurations of a Turing machine. The fact that $(s', \overline{j}', \overline{c}')$ is a successor configuration of $(s, \overline{j}, \overline{c})$ is denoted $(s, \overline{j}, \overline{c}) \triangle (s', \overline{j}', \overline{c}')$. Here $\triangle$ is a binary relation on the set of all configurations and forms a directed graph. We refer to this graph as the configuration graph of $\mathbb{M}$.

Any computation of the Turing machine $\mathbb{M}$ is a directed path in this graph configuration. A run of $\mathbb{M}$ is on $x$, if the input of the computation is $x$, i.e. the starting configuration is on $x$. A computation is complete if it ends in a halting configuration. If in a complete computation of the Turing machine $\mathbb{M}$ on $x$, the cell content is given by $\overline{c} = c_1...c_k$, then the output of the computation is the binary string $c_k(1)...c_k(j)$, where $j + 1$ is the least with $c_k(j+1) = \square$. In particular a output is defined only for complete computations. The Output of a complete computation is the empty string. In the case described above it is 0. The Turing machine $\mathbb{M}$ computes the partial function, that sends $x$ to the output of a complete run of $\mathbb{M}$ on x and remains undefined if no complete run of $\mathbb{M}$ on x exists. A complete run on $x$ is accepting this, if $c_k(1) = 1$ and rejecting it, if $c_k(1) = 0$. Equivalently, we say that $\mathbb{M}$ accepts respectively rejects x.

**Definition 31.** Let $Q$ be a problem, which we have identified as a set of binary strings,

$$Q \subseteq \{0, 1\}^\tau = \cup^n_{n \in \mathbb{N}} \{0, 1\}.$$

(1) A Turing machine $\mathbb{M}$ is said to decide the problem Q if it accepts every $x \in Q$ and rejects every $x \notin Q$. The problem Q is said to be decidable, if there is a Turing machine $\mathbb{M}$ which decides it.

(2) A Turing machine $\mathbb{M}$ is said to accept the problem $Q$ if

$$Q = \{x \in \{0,1\}^{\tau} : \mathbb{M} \text{ accepts } x\}$$

So from this one can see that the class of Turing computable functions coincides with the class of general recursive functions and so the class of computable functions. If we recall that a problem $Q$ is decidable iff its characteristic functions is computable, which is exactly the first part of the definition given above.

7.1. **Robustness.**

**Definition 32.** Consider a Turing machine with two special tapes:

(1) An input tape, where the input is written. The content of the cells in the input tape are not changed and the head does not move to the right of the last input cell.
(2) On the output tape the machine produces output of a computation. The head is only allowed to move from left to right. We say that the Turing machine has and input and output tape.

The goal is to show how any $k$- tape Turing machine can be simulated by a $(k+2)-tape$ Turing machine, with input and output tapes. It is necessary to understand how this exactly works. First one copies the input from the first tape, to the second tape. Then next step is to run the $k$-tape Turing machine program on tapes $2, ..., k+1$. Finally the content of the $k+1$-tape is copied to the $k+2$-tape.

Furthermore one can consider a Turing machine defined by a larger alphabet $\Sigma$. Note that $|\Sigma| \leq 2^{[log_2|\Sigma|]}$. An arbitrary $\Sigma$-Turing machine can be simulated by the usual Turing

machine on the alphabet $\{0,1\}$ in which each element of $\Sigma$ is encoded by a string of $0,1$, say of length $log|\Sigma|$ . When the $\Sigma$-TM writes one symbol $\diamondsuit$ and moves to the right, the corresponding standard machine writes the $log|\Sigma|$ symbols encoding $\diamondsuit$ and moves $2log|\Sigma|$ - many cells to the right. Thus every symbol is regarded, even the blank symbol is substituted. So really every $k-tape$ TM can be simulated by a single-tape TM. So the idea is, given a $k$- tape TM $\mathbb{M}$ and the single-tape TM $\mathbb{S}$, $\mathbb{S}$ will use a new symbol #. Then the plan is that $\mathbb{S}$, will write consecutively the content of the $k$-tape on its single tape, separating them with the symbol #. In addition $\mathbb{S}$ must keep track of the location of each virtual head. For example by writing "hat" over the corresponding symbols.

Given an input $x_1, ..., x_n$:

(1) $\mathbb{S}$ puts its tape in a format representing the $k-$tapes of $\mathbb{M}$,

e.g $\#\hat{x_1}x_2...x_n\#\hat{\square}\#\hat{\square}\#\hat{\square}...\#$

(2) To simulate a move, $\mathbb{S}$ scans its tape from the first # and finds out the symbol under the virtual heads. Then it goes over the tape a second time to execute the transition function of $\mathbb{M}$.

(3) If it happens that during the calculation a virtual head is placed over #, this means that the original head was over $\square$. Then $\mathbb{S}$ substitutes # with $\square$ and shifts everything relevant to the right by a magnitude of 1 cell.

7.2. **Computation Tables.** Before giving a precise definition of what a computation table of a TM is, let's look at an example of a transition function. Consider a 2-tape TM $\mathbb{M} = \mathbb{S}, \delta$ that reverses its input entry. For example if 00110 is the input, then the output is 01100. Here $\mathbb{M}$ is

(1) Set of states: $\mathbb{S} = \{S_{start}, S_{halt}, s_r, s_l\}$

(2) Transition function: $\delta = (S_{start}, \triangle\triangle) = (S_r, \triangle\triangle 10)$ (1 meaning move to the right, 0 meaning don't move)

(3) Read the input: $\delta = (S_r, b\triangle) = (s_r, b \triangle -11)$, $b \in \{0, 1\}$

(4) When the input is read, start moving back: $\delta = (S_r, \square\triangle) = (s_l, \square \triangle -11)$

(5) While reading backwards the first tape, copy it on the second tape:

$$\delta = (S_l, b\square) = (s_l, bb, -11),\ b \in \{0, 1\}$$

(6) Halting: $\delta = (S_l, \triangle\square) = (s_{halt}, \triangle\square 00)$.

After looking at this example here is the exact definition of a computation table:

**Definition 33.** Let $\mathbb{M} = (\mathbb{S}, \delta\}$ be a single-tape TM, with $x \in \{0, 1\}^{\mathbb{S}}$ and $t \geq 1$. A computation table of $\mathbb{M}$ on $x$ up to step $t$ is a matrix $\{T_{i,j}\}_{i\in[t]}, j \leq \hat{t}\}$ where

$$\hat{t} := max\{|x|, t\} + 1 \text{ and } \forall i, j : T_{i,j} \in \{0, 1, \square\triangle\} \cup \{0, 1, \square, \triangle\} \times \mathbb{S}$$

More precisely, if $(s_i, j_i, c_i)$ is the $i$-th configuration of the run if $\mathbb{M}$ on x, then

$$T_{ij} = \begin{cases} c_i(j) & \text{if } j \neq i_j \\ (c_i(j), s_i) & \text{if } j = i_j \end{cases}$$

The last column of a computation table contains only blanks.

## 8. Memory Complexity

Now after showing what a TM is respectively showing how it operates, the goal is to look at the second core limitation of computation, namely memory complexity. The theoretical limitation, as shown above, lies in the problem of non-computable functions. In the following section the issue of time complexity and space complexity will be investigated.

### 8.1. **Time complexity.**

**Definition 34.** Let $\mathbb{M}$ be a TM and $f : \mathbb{M} \to \mathbb{M}$ a function. $\mathbb{M}$ is said to be $f$-time bounded, if for every $x \in \{0, 1\}^{\mathbb{S}}$ ,there is a complete run of $\mathbb{M}$ on $x$ that has length of

most $f(|x|)$, where $|x|$ denotes the length of $x$. We say that $\mathbb{M}$ is a $f(n)$-time TM, where $n = |x|$. Note that the upper bound we are given depends only $|x|$, but not $x$ itself.

**Definition 35.** Let $g, f : \mathbb{N} \to \mathbb{R}^+$. We say that:

(1) $f(n) \leq O(g(n))$ if $\exists c, n_0 \in \mathbb{N}$ such that $\forall n \geq n_0 f(n) \leq cg(n)$. $g(n)$ is said to be an asymptotic upper bound to $f(n)$.
(2) $f(n) \geq \Omega$ if $g \leq O(f)$.
(3) $f = \Theta$ if $f \leq O(g)$ and $g \leq O(f)$.

So for example if we are given a function $f(n) = 6n^3 + 4n^2 + 7n + 5$, then $f_1(n) = O(n^3)$. But also $f_1(n) = O(n^4)$. However $f_1(n) \neq O(n^2)$.

Another example would be $x = log_2 n$. Recall that $log_B n = \frac{log_2 n}{log_2 b}$ and so $x = (log_2 b) log_B n$. Thus if $f(n) = O(log_2 n)$, then also $f(n) = O(log_6)$ and we don't have to specify the base $b$. Interestingly both $2^{O(log n)} : n = 2^{log_2 n}$ and $n^c = 2^{c log_2 n}$ have the same upper bound, since $f(n) = 2^{clog_2} = n^c$ for some c. Finally even $n^{O(1)}$ is the same as $n^c$ for some constant $c$.

**Definition 36.** Let $g, f : \mathbb{N} \to \mathbb{R}^+$. We say that:

(1) $fn \leq O(g(n))$ (we say, f is asymptotically smaller then $g$) if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$. That is, if $\forall c > 0$ there $\exists n_0 \in \mathbb{N} \forall n \geq f(n) < cgn(n)$.
(2) $f \geq \omega$ iff $g \leq O(f)$.

Note that $f(n)$ can never be $O(f(n))$.

**Definition 37.** Let $g, f : \mathbb{N} \to \mathbb{R}^+$. By $TIME(t)$ we denote the class of all problem $Q$, so that $\exists c \in \mathbb{N}$ and some $(ct + c)$ bounded TM $\mathbb{M}$, that decides $Q$.

We also say that $\mathbb{M}$ is a $(ct + c)$-time TM.

The following three classes will be interesting for the context of this thesis:

(1) $\mathbb{P} := \bigcup TIME(n^c)$ , called polynomial time

(2) $\mathbb{E} := \bigcup TIME(2^{cn})$ , called simple exponential time

(3) $\mathbb{EXP} := \bigcup TIME(2^{n^c})$ , called exponential time

A natural question that arises from this definition is how to determine, how long it takes a single-tape TM to decide a given decidable set A.

**Theorem 38.** *Suppose* $f : \{0,1\}^{\mathbb{S}} \to \{0,1\}^{\mathbb{S}}, t : \mathbb{N} \to \mathbb{N}$. *If* $f$ *can be computed by a* $t(n)$*-time TM* $\mathbb{M}$ *with* $k$ *tapes, where* $k > 0$*, then* $f$ *can be computed by a single-tape TM* $\mathbb{S}$ *which runs in time* $O(t^2(n))$.

*Proof.* The idea is to show that simulating each step of the multi-tape TM uses at most $O(t(n))$ steps on the single-tape machine.

Let $\mathbb{M}$ be a $k$-tape TM that runs in time $t(n)$. We can construct a single tape TM $\mathbb{S}$, that runs in time $O(t^2(n))$ and simulates $\mathbb{M}$. The $\mathbb{M}$-tapes are stacked consecutively with the positions of $\mathbb{M}^s$ heads marked appropriately.

(1) To simulate one step, $\mathbb{S}$ makes two passes over the active part of its tape: the first to obtain the information and the second one to carry it out. Note that each active portion of $\mathbb{M}$ has length of at most $t(n)$, because $\mathbb{M}$ uses $t(n)$ tape cells in $t(n)$ steps. Thus to scan its active part $\mathbb{S}$ uses $O(t(n))$- steps. In addition there might be up to $k$ rightwards shifts, each using no more than $O(t(n)$-time. Thus simulating one step of $\mathbb{M}$, $\mathbb{S}$ uses no more than $O(t(n))$-time.

(2) The computation by $\mathbb{M}$ ends in $t(n)$ steps, which gives a total of $t(n)O(t(n)) = O(t^2(n))$ steps.

□

To give an example, consider following problem called Reachability. Given a directed graph $G = (V, E)$ and vertices $x, y$, is there a path from $x$ to $y$ in $G$? Now the idea is to find the set of vertices $x_0$, which is closed under $E$-successors and contains $x$. Then check

if $y \in X_0$. Here is an algorithm how to find the smallest such $X_0$, meaning the smallest set containing $x, y$:

(1) Take $X_0 = \{X\}, Y = \emptyset$.
(2) Substitute $X_0$ with $X_0 \cup y$ (note that this can be done in $O(n)steps$ )
(3) For all edges $(x_i, x_j) \in E$ , if $x_i \in X_0$ substitute $Y$ with $Y$ $cup\{x_j\}$ (note that this can be done in $O(n)$-steps, because there are $n^2$ edges)
(4) If $Y \not\subseteq X_0$, then go back to step 2. (note that this steps takes ($O(n)$-steps)
(5) If $y \in X_0$ accept, otherwise reject.

Thus we can describe the entire graph with a binary string of length $O(n^2)$. Since step 2,3,4 can be repeated at most $n$-times an upper bound of $nO(n^2) = O(n^3)$ is obtained and we showed that the problem of reachability can be solved in polynomial time $\mathbb{P}$.

8.2. **Polynomial time universal TM.** A TM $\mathbb{M}$ is uniquely determined by its transition functions. The transition functions is a finite set and so can be encoded by a binary string. Note that there are many such encodings. Furthermore, we can find an encoding of $\mathbb{M}$ with a binary string $\lceil \mathbb{M} \rceil$ in such a way that: for each $(s, a) \in dom(\delta)$, we can compute $\delta(s, a)$ in time $O(\lceil \mathbb{M} \rceil)$.

**Theorem 39.** *There is a UTM (universal TM) such that given $\lceil \mathbb{M} \rceil$, where $\mathbb{M}$ is a single-tape TM, $x \in \{0, 1\}$ and $1^t = 11...1$ for some $t \in \mathbb{N}$, the machine computes in quadratic time (so polynomial time) the $t$-th row of the computation table of $\mathbb{M}$ on $x$ up to step $t$.*

*Proof.* Each entry of the table, i.e. $T_{ij}$, can be encoded by a binary string of length $O(logs)$, where is anything but equal to the states of $\mathbb{M}$ and so a row can be encoded by an binary string of length $O(logs \times max\{t, |x|\})$.

There are $t$-may rows. In addition, by hypothesis on the encoding function $\lceil \rceil$ for $\mathbb{M}$, given any configuration $(s, j, c)$, a successor can be decoded from $\lceil \mathbb{M} \rceil$ in linear time. Thus, having $\lceil \mathbb{M} \rceil$ the universal time machine can compute the next row of $\mathbb{M}$ $n$ -time linear in

$|\lceil \mathbb{M} \rceil|$. Therefore the total time of the computation is $O(t^2|x||\lceil \mathbb{M} \rceil|$ and since $|\lceil \mathbb{M} \rceil|$ and $|x|$ are treated as constants, which is the same as $O(t^2)$.

□

8.3. **Non-deterministic Turing machines.** So far only deterministic TM where discussed. Another very important model for computations are nondeterministic Turing machines. In contrast to deterministic TMs, that at every step of the configuration have only one option to proceed, which is fully determined by there initial state respectively there transition function, non deterministic TMs have multiple options to choose from at every given step of the computation. The $P = NP$-problem is exactly the question, if every computation that can be carried out by a nondeterministic TM can, in general, be carried out by a deterministic TM.

**Definition 40.** Nondeterministic Turing machines

(1) A non deterministic TM is a decider, if all branches halt in all outputs.

(2) Let $\mathbb{M}$ be a decider. The running time $t : \mathbb{M} \to \mathbb{M}$ of $\mathbb{M}$ is the maximal number of step $t(n)$ along a branch of the computation tree of $\mathbb{M}$ in an input of length $n$. We say also that $\mathbb{M}$ is of time $t(n)$.

**Definition 41.** A function $f : \mathbb{N} \to \mathbb{N}$ is time constructable iff $\forall t(n) \geq n$ and the function $x \to bin(t(|x|)$ is computable in time $O(|t(x)|)$.

**Theorem 42.** *Let $t(n) \geq n$ be a time constructable function. Let $\mathbb{M}$ be a single-tape, non deterministic TM running in time $t(n)$. Then there is a deterministic TM $\mathbb{M}'$ running in time $2^{O(t(n))}$ such that, for every problem $Q \subseteq \{0,1\}, \mathbb{M}$ decides $Q$ iff $\mathbb{M}'$ decides $Q$. We say that $\mathbb{M}$ and $\mathbb{M}'$ are equivalent.*

*Proof.* The following can be considered more as an outline of a proof rather then a proof. Still it highlights the most important ideas, satisfying curiosity sufficiently for the time being. A full proof can be found in [3].

First simulate $\mathbb{M}$ on a 3-tape deterministic TM $\mathbb{M}_s$. The first tape is used to keep the input. The second tape is a simulation tape. The machine contains the address of a path through the computation tree of $\mathbb{M}$.

For each $(s, r) \in \mathbb{S} \times \Sigma$ let $\delta_{s,\sigma} = (\triangle(s, \sigma))$ and let

$$b = max\{\delta_{s\sigma} : (s, \sigma) \in \mathbb{S} \times \Sigma\}.$$

The computation tree of $\mathbb{M}$ is at most $b$-branching, i.e. every configuration has at most $b$ many successors. If we enumerate all nodes of the tree, then a computation path through the tree is determined by a sequence of natural numbers of length smaller than $t(n)$. Those can be lexiographically ordered. The total total number of nodes is,

$$\sum_{i=0}^{t(n)} b^i$$

which is smaller than $2b^{t(n)}$. That means that the number of nodes is smaller than $O(b^{t(n)})$. To reach $b^t(n)$ from the root of c, we need no more than $t(n)$ steps. Thus $\mathbb{M}_1$ is running in time $O(t(n)b^{t(n)}) = 2^{O(t(n))}$. Now take a single tape TM $\mathbb{M}'$ equivalent to $\mathbb{M}_1$, which runs in time

$$2^{O(t(n))2} = 2^{O(2t(n))} = 2^{O(t(n))}$$

.

□

**Definition 43.** Let $t : \mathbb{N} \to \mathbb{N}$. Then $\mathbb{N}TIME(t)$ denotes the class of all problems $Q$ for which there is a $c \in \mathbb{N}$ and a nondeterministic TM $\mathbb{N}$ running in time $ct + c$, which accepts $Q$.

(1) $\mathbb{NE} = \bigcup_{k\in\mathbb{N}} \mathbb{N}TIME(2^{kn})$ is called simply exponential time.
(2) $\mathbb{NEXP} = \bigcup_{k\in\mathbb{N}} \mathbb{N}TIME(2^{nk})$ is called nondeterministic exponential time.
(3) $\mathbb{NP} = \bigcup_{k\in\mathbb{N}}(n^k)$ is called nondeterministic polynomial time.

It can be shown (and later will be in this thesis) that the class $\mathbb{P}$, which is the class of all functions a deterministic TM can compute in polynomial time, is a subset of $\mathbb{NP}$ and later being a subset of $\mathbb{EXP}$ ,

$$\mathbb{P} \subseteq \mathbb{NP} \subseteq \mathbb{EXP}$$

Furthermore we know that $\mathbb{P} \subsetneq \mathbb{EXP}$, which will not be proven, since this would go beyond the scope of this thesis. A proof is given for example in [3]. The really interesting question is whether or not $\mathbb{P} \subsetneq \mathbb{NP}$, often referred to as the $\mathbb{P} \neq \mathbb{NP}$-problem. If $\mathbb{P}$ is a real subset of $\mathbb{NP}$ the later statement would be true.

Another way to look at the $\mathbb{P} \neq \mathbb{NP}$-problem concerns the computation tree of a nondeterministic TM. When we go over the branches of such computation tree, an exhaustive search is being executed, which leads to an algorithm, that requires exponential time to be executed. Now given a solution in exponential time, does there exist a more efficient solution in polynomial time. If that was the case, then $\mathbb{P} = \mathbb{NP}$ would follow, since every problem carried out by a nondeterministic TM could be reduced to a problem solvable in $\mathbb{P}$. So before moving on it can be useful and interesting to look at another respectively alternative definition of $\mathbb{NP}$.

**Definition 44.** Polynomial bounds

Let $Re \subseteq \{0,1\}^{\mathbb{S}} \times \{0,1\}^{\mathbb{S}}$ be a relation.

(1) $R$ is said to be polynomial bounded if there exists a polynomial $p(x)$ and for all $(x,y) \in R$, we have $|y| \leq p(x)$.

(2) $R$ is said to be in $\mathbb{P}$ if $\{(x,y) : (x,y) \in R\} \in \mathbb{P}$.

For example let $\mathbb{M}$ be a NTM which runs in polynomial time (say $\mathbb{M}$ a decider). Let $R = \{(x,y) : y \text{ is a branch through the computation tree of } \mathbb{N} \text{ on } x\}$. Then $R$ is said to be in $\mathbb{P}$ if $\{(x,y) : (x,y) \in R\} \in \mathbb{P}$.

**Theorem 45.** *A problem $Q \subseteq \{0,1\}^{\mathbb{S}}$ is in $\mathbb{NP}$ iff there is a polynomialy bounded relation $R$, which is in $\mathbb{P}$ and $Q = domR = \{x : \exists y(x,y) \in R$*

*Proof.* ($\Leftarrow$) Let $R$ be a polynomially bounded relation and let $\mathbb{M}$ be a TM running in time $n^k$ deciding $R$. Furthermore let $Q = dom(R)$ We will construct a NTM $\mathbb{M}_1$ deciding $Q$ as follows: Since $R$ is polynomially bounded, there is a polynomial $p(x)$ such that $\forall(x,y) \in R, |y| \leq p(|x|)$. We get $p(|x|) \leq O(n^k)$ where $n = |x|$. Now, given an input $x \in \{0,1\}^{\mathbb{S}}$, nondeterministically choose a bitstring $y$ of length $n^k$ and run $\mathbb{M}$ on $(x,y)$). If $\mathbb{M}$ accepts, then $\mathbb{M}_1$ should accept as well. If $\mathbb{M}$ rejects, then $\mathbb{M}_1$ should reject as well.

($\Rightarrow$) Let $\mathbb{M}$ be a NTM deciding $Q$ and running in time $p(x)$. If $x \in Q$, fix a branch $y_x$ through the computation tree of $\mathbb{M}$ on $x$, which yields an accepting configuration. Then $|y_x| \leq p(|x|)$.

Now let $R = \{(x, y_x) : x \in Q\}$. Thus $Q = dom(R)$ and $R$ is polynomially bounded. To see $R$ is in $\mathbb{P}$, consider a deterministic TM $\mathbb{M}_1$, which works as follows: On input $\langle x, y\rangle$ where $|y| \leq p(|x|)$, $\mathbb{M}_1$ interprets $y$ as a branch $b(y)$ through the computation tree of $\mathbb{M}$ and simulates $\mathbb{M}$ on $x$ along $b(y)$. If $b(y)$ is not a branch, or $\mathbb{M}$ rejects, then $\mathbb{M}_1$ rejects. Otherwise $\mathbb{M}_1$ accepts. □

8.4. **Space complexity.** After describing time complexity in terms of timely boundaries a certain computation takes to run, the next step is to introduces space complexity, as another practical limit to computation. The question is how much storage space is necessary for the run of a computation depending on the size of the input respectively how fast the demand of storage increases compared to the increase of the input.

**Definition 46.** Let $\mathbb{M}$ be a TM (deterministic or non deterministic) with an input, an output and $k$-working tapes.

(1) A configuration of $\mathbb{M}$ is a triple $(s, ij\overline{j}, j, c\overline{c})$ where $s$ is the current state, $i$ is the location of the on the input tape, $\overline{j} = j_1...j_k$ are the contents of the working tapes and $c$ is the content of the output tape.

(2) Let $\mathbb{S} \in \mathbb{N}$. A configuration of $\mathbb{M}$ is $\mathbb{S}$-small if $j_1...j_k \in \mathbb{S}$ and on all work tapes, all cells with numbers $> \mathbb{M}$ are blank.

(3) Let $s : \mathbb{M} \to \mathbb{M}$. We say that $\mathbb{M}$ is $s(|x|)$-bounded, if for every $x \in \{0,1\}$, every configuration of $\mathbb{M}$, reachable in the configuration graph from the starting configuration of $\mathbb{M}$ on $x$,is ($s(|x|)$-small.

For example on could consider the configuration table of a single-tape TM, that apart from the input row never has more than $s(|x|)$-many columns. Similar two timely bounds, space bounds can be introduced, to classify the demand of storage an algorithm uses.

**Definition 47.** Space bounds

(1) $SPACE(s)$ is the set of problem $Q$, such that $\exists c \in \mathbb{M}$ and $\exists(cs+c)$-space bounded TM $\mathbb{M}$ deciding $Q$.

(2) $\mathbb{N}SPACE(s)$ is the set of problems $Q$ that $\exists c \in \mathbb{N}$ and $\exists(cs+c)$-space bounded $\mathbb{N}$ $\mathbb{M}$ deciding $Q$.

(3) $\mathbb{L} := SAPCE(logn)$, $\mathbb{P}SPACE := \bigcup_{k\mathbb{M}}(n^k)$

**Definition 48.** A function $s : \mathbb{N} \to \mathbb{M}$ is space-computable if

(1) $s(n) \geq logn \, \forall n$

(2) The function $x \to bin(s(|x|))$ is computable by a $O(s)$- space bounded TM.

Remark: Let $\mathbb{M}$ be a deterministic or nondeterministic TM with $k$-working tapes. An $\mathbb{S}$-small configuration of $\mathbb{M}$ on input x, where $|x| = n$, can be encoded by an binary string of length $O(log(|\lceil\mathbb{M}\rceil| + logn + klog\mathbb{S} + k\mathbb{S})$. In order to encode this, we need to encode following things:

(1) A state of $\mathbb{M}$, which can be encoded by a string of length $log(|\lceil\mathbb{M}\rceil|)$.

(2) The position of the reading-head, which can be achieved by a string of length $logn(n = |x|)$

(3) The position of the work-heads, which can be encoded by a string of length $k\mathbb{S}$, since we have $k$ many strings each of length $log\mathbb{S}$.

(4) The contents of the work tapes, which can be achieved by a string of length $k\mathbb{S}$, since the configuration is $\mathbb{S}$-small, so each tapes contains a binary string of length at most $\mathbb{S}$.

**Theorem 49.** *Space Hierarchy*

*Let $s$ be space-constructable and let $s' \leq O(s)$. Then*

$$SPACE(S) \backslash SPACE(s') \neq \emptyset.$$

*Proof.* Consider the problem $\mathbb{P}_S = \{\lceil \mathbb{M} \rceil : \mathbb{M}$ is a $k$-tape TM such that,

(1) during the computation in $\lceil \mathbb{M} \rceil$ visits $\lfloor \frac{s(|\lceil M \rceil|)}{k} \rfloor$, or

(2) $\mathbb{M}$ does not accept the input $\lceil M \rceil$ $\}$.

We will show that

$$\mathbb{P}_S \in SAPCE(s), \text{ but } \mathbb{P}_S \notin SAPCE(s').$$

First, we construct a TM $\mathbb{M}'$, which decides $\mathbb{P}_S$ and is $s$-bounded. On input $\lceil \mathbb{M} \rceil$, the TM $\mathbb{M}'$ works exactly as $\mathbb{M}$ , but whenever for some $j_l, l \in [k]$, the transition function on $\mathbb{M}$ wants to move the $l$-th working head on location

$$\lfloor \frac{s(|\lceil \mathbb{M} \rceil|)}{k} \rfloor,$$

$\mathbb{M}'$ terminates the computation in the accepting state. If this does not happen and $\mathbb{M}$ does not accept $\lceil \mathbb{M} \rceil$, then $\mathbb{M}'$ has to accept $\lceil \mathbb{M} \rceil$. But if $\mathbb{M}$ rejects $\lceil \mathbb{M} \rceil$, then $\mathbb{M}'$ accepts $\lceil \mathbb{M} \rceil$ (in case the first option does not happen). Moreover, every time a new configuration is computed, the old one is deleted. We always keep the starting configuration. In order to find out how much space $\mathbb{M}'$ needs, we need to carry out the following steps:

(1) Encoding the state can be achieved in $O(log(\lceil \mathbb{M} \rceil)$- many bits. Suppose $\mathbb{M}$ has $l$-many states. Thus all states of $\mathbb{M}$ can be encoded by a binary string of length $log(l)$. Recall that $\lceil \mathbb{M} \rceil$ encodes all states of $\mathbb{M}$ and so $l \leq (|\lceil \mathbb{M} \rceil|$.

(2) Encoding the position of the reading head can be achieved in the following way: The available positions are $\triangle, ..., |\lceil \mathbb{M} \rceil|$. Thus we can encode the position with a binary sting of length $O(log(|\lceil \mathbb{M} \rceil|)$.

(3) The content of the work tapes up to cell

$$\frac{s(|\lceil \mathbb{M} \rceil|)}{k}$$

can be encoded this way: Since there are $k$- many tapes, this can be achieved in

$$k\frac{s(|\lceil \mathbb{M} \rceil|)}{k} = s(|\lceil \mathbb{M} \rceil|)$$

-many bits.

Therefore a working configuration of $\mathbb{M}'$ can be encoded by no more than

$$O(log|\lceil \mathbb{M} \rceil| + log(\lceil \mathbb{M} \rceil + s(|\lceil \mathbb{M} \rceil|)) \leq O(s(|\lceil \mathbb{M} \rceil|)) - \text{space},$$

where we used that $log|\lceil \mathbb{M} \rceil| \leq s(|\lceil \mathbb{M} \rceil|)$ since $s$ is space-constructable. Thus, there is a constant $c$ such that a configuration of $\mathbb{M}'$ can be encoded by no more than $cs(|\lceil \mathbb{M} \rceil|)$-many bits. Thus we are in particular identifying configurations of $\mathbb{M}'$ on $\lceil \mathbb{M} \rceil$ with binary strings of length $cs(|\lceil \mathbb{M} \rceil|)$. Note however, that there are only

$$2^{cs(|\lceil \mathbb{M} \rceil|)}$$

- many possibilities for such a string. Thus, in more than $2^{cs(|\lceil \mathbb{M} \rceil|)}$-many steps, the program will enter a configuration it already was in and so will enter a loop. To avoid this, we count the number of configurations and guarantee that $\mathbb{M}'$ enters a rejecting state, whenever this

number reaches $2^{cs(|\lceil\mathbb{M}\rceil|)}$. For this we need at most

$$log(2^{cs(|\lceil\mathbb{M}\rceil|)}) = cs(|\lceil\mathbb{M}\rceil|)$$

- many bits. Now, $\mathbb{M}'$ clearly decides $\mathbb{P}_s$ and $\mathbb{M}'$ is $s$-bounded. Thus

$$\mathbb{P}_\sim \in \mathbb{SPACE}(\sim)$$

. The next step is to show that $\mathbb{P}_\sim \notin SPACE(s)$ whenever $s' \leq O(s)$. To achieve this we fix $s' \leq O(s)$ and suppose $\mathbb{P}_\sim \in SPACE(s)$. Thus $\exists$ TM $\mathbb{M}_*$ which is $(cs' + c)$- space bounded and decides $\mathbb{P}_\sim$. Let $k$ equal the number of work tapes of $\mathbb{M}_*$. By definition, since $s' \leq O(s)$, $\exists n_0 \in \mathbb{N}\, \forall n \geq n_0$ with

$$cs'(n) + c < \lfloor \frac{s(n)}{k} \rfloor$$

where $|\lceil\mathbb{M}_*\rceil| \geq n_0$, indeed. If not, simply add redundant states of $\mathbb{M}_*$. If $\mathbb{M}_*$ accepts $|\lceil\mathbb{M}_*\rceil|$.

If $\mathbb{M}_*$ does not accept $\lceil\mathbb{M}_*\rceil$, then by definition of $\mathbb{P}_\sim$, $\lceil\mathbb{M}_*\rceil \in \mathbb{P}_\sim$. However, by assumption $\mathbb{M}_*$ decides $\mathbb{P}_\sim$, which means that the computation with input $\lceil\mathbb{M}_*\rceil$ terminates in and accepting state, which is a contradiction.

Then, it must be the case that $\mathbb{M}_*$ does accept $\lceil\mathbb{M}_*\rceil$. Since $\mathbb{M}_*$ decides $\mathbb{P}_\sim$, this means that $\lceil\mathbb{M}_*\rceil \in \mathbb{P}_\sim$. By definition of $\mathbb{P}_\sim$ this implies that during the computation on $\lceil\mathbb{M}_*\rceil$ one of the heads of $\mathbb{M}_*$ visits location

$$\lfloor \frac{s(|\lceil\mathbb{M}_{\leftarrow}\rceil|}{k} \rfloor$$

. This is a contradiction to

$$cs'(n) + c < \lfloor \frac{s(n)}{k} \rfloor$$

Indeed, since $\mathbb{M}_*$ is $cs' + c$-space bounded and $|\lceil\mathbb{M}_*\rceil| \geq n_0$, in any configuration of $\mathbb{M}_*$, $\mathbb{M}_*$ does not visit locations $\geq cs' + c$. Therefore $\mathbb{P}_\sim \notin Space(s')$

□

So the Space hierarchy theorem assures us that a TM (deterministic or not) is asymptotically better if the space boundaries are loosened, see [12]. In this sense it is even stronger than the other hierarchy theorem, namely the time hierarchy theorem, which will not be demonstrated in this thesis. The main point is, that the intuition is right, that with more spacial and timely resources computational power increases significantly.

## 9. Relations between complexity classes

Now that we established different complexity classes it will be interesting to investigate relations between. The most prominent question of course refers to the $\mathbb{P} \neq \mathbb{NP}$- problem: Is the class of functions a deterministic TM can compute in polynomial time the same as a nondeterministic TM can in the same time? Of course this thesis can not contain a decision of this problem it shall outline a path to where the problem can be addressed from properly.

**Definition 50.** A function $f : \mathbb{N} \to \mathbb{N}$ is a proper complexity function, if $f$ is nondecreasing and there is a $k$-string TM $\mathbb{M}_{\mho}$ with an input and and output tape such that in input $x$,$\mathbb{M}_{\mho}$ computes a designated string of length $f(|x|)$, stops in more than $O(|x|+f(|x|)$-many steps and uses more than $O(f(|x|)$ space. Defining time or space complexity always refers to proper complexity functions.

**Theorem 51.** *Let $f(n)$ be a proper complexity function. Then*

*(1)* $SPACE(f(n)) \subseteq NSPACE(f(n))$ *and* $TIME(f(n)) \subseteq NTIME(f(n))$

*(2)* $NTIME(f(n)) \subseteq SPACE(f(n))$

*(3)* $NSPACE(f(n)) \subseteq TIME(k^{logn+f(n)})$.

*Proof.* In order to proof the theorem stated above, the following steps will be necessary:

(1) Any deterministic TM is also non-deterministic.

(2) Let $Q \in NTIME(f(n))$, Thus $Q \subseteq \{0,1\}$ and there is a nondeterministic TM $\mathbb{M}$, that decides $Q$ in time $f(n)$. The computation of $\mathbb{M}$ can be simulated by a deterministic a deterministic TM $\mathbb{M}'$, which in fact has to simulate every branch of the computation tree of $\mathbb{M}$. Since every branch of $\mathbb{M}$ has length at most $f(n)$ (because $\mathbb{M}$ is a decider), $\mathbb{M}'$ can simulate this computation in space $f(n)$ (each step along the branch takes 1 cell). Even though there are exponentially many branches, $\mathbb{M}'$ can reuse the space to simulate each of them. Even if the computation tree on an input of length $n$ of $\mathbb{M}$ has $2^{f(n)}$-many branches, to keep track of the branches, $\mathbb{M}'$ needs only

$$O(log 2^{f(n)} = O(f(n))$$

much space. Thus $\mathbb{M}'$ works in $f(n)$-space and is therefore $f(n)$-space bounded.

(3) Let $Q \in NSPACE(f(n))$ and let $\mathbb{M}$ be a $k$-tape NTM, which works in space $f(n)$ and decides $Q$. Thus at any step of the computation on input $x$, $|x|$, all active strings are of length at most $f(n)$. How many configuration are there in input $x$, $|x| = n$?

(1) $|S|$-many choices for a state

(2) $n+1$-many choices for the position of the first head

(3) $(k-2)|\Sigma|^{f(n)}$-many choices for the contents of the tape.

Thus the total number of configurations is

$$\leq (nc_1^{f(n)} \leq c_1^{logn+f(n)})$$

since $n = c_1^{log_{c1}n}$. Now considering the configuration graph of $\mathbb{M}$ on input $x$:

$$G(\mathbb{M}, x) = \langle \text{Configuration, SuccesorRelation} \rangle.$$

Since $\mathbb{M}$ decides $Q$ we have that $x \in Q$ iff there exists a path $P$ in $G(\mathbb{M}, x)$ from the starting configuration on $x$ to an accepting configuration. Thus, we reduced the problem

to the reachability problem of a graph, with $c_1^{3logn+f(n)}$ which as we saw above can be solved in $O(c_1^{3logn+f(n)})$-deterministic time. Thus

$$Q \in TIME(k^{logn+f(n)})$$

□

9.1. **Reachability Method.** In general a lot of times it can be helpful to reduce a computational problem to a question of reachability: can a certain state be reached given a set of initial conditions like in case of the TM. The theorem below follows [14]

**Theorem 52.** *The Theorem of Savitch*

$$PATH \in SPACE(log^2 n)$$

*Proof.* Let $G$ be a graph, $V[G] = [n]$. For $x, y \in V(G)$ and $n \geq 0$, let $PATH(x, y, i)$ be a predicate, which is true iff there exists a path from $x$ to $y$ in $G$ with a length of at most $2^i$. Thus, to determine if there is a path between 2 nodes of $G$, it is sufficient to find out if $PATH(x, y, \lceil logn \rceil)$ holds.

We will define a TM $\mathbb{M}$ with 3-tapes (input tape, 2 working tapes), which decides $PATH(x, y, i)$ in space $log^2 n$. First, the adjacency matrix of of $G$ will be given in the input tape of $\mathbb{M}$. At each configuration, the first working tape will contain $bin(x), bin(y), bin(i)$ and at most $\lceil logn \rceil$ many triples of the form $(x, y, i)$. Since there are only $n$-many nodes, this takes at most $\lceil logn \rceil 3logn = O(log^2 n)$ many bits. The machine functions as follows

(1) if $i = 0$, then check if $PATH(x, y, 0)$ by checking if $x = y$ or $x, y$ are adjacent.

(2) if $i \geq 1$, then compute $PATH(x, y, i)$ using the recursion below:

(3) For all vertius $z$ check if $PATH(x, z, i - 1)$ and $PATH(z, y, i - 1)$.

Here is the algorithm for $PATH(x, y, i)$: Write $(x, y, i)$ on tape 1. The idea behind this step is to exhaustively check $*$? for each $z$. Then z is chosen and $(x, z, i-1)$ is written on the tape 1 and $PATH(x, z, i-1)$ is checked. If the TM rejects the input, erase $(x, z, i-1)$ from tape 1 and pick another $z$. If the machine accepts the input, substitute $(x, z, i-1)$ with $(z, y, i-1)$ on tape 1 and check if $PATH(z, y, i-1)$. If the TM now again rejects the input, erase $(z, y, i-1)$ from tape 1 and pick another $z$. If the input is accepted, return to substituting.

We start the program with $(x, y, \lceil logn \rceil)$ written on tape 1. Note that if $PATH(x, y, i)$ then $\exists z$ with $PATH(x, z, i-1)$ and $PATH(z, y, i-1)$. At each moment of the computation, the first work tape contains at most $\lceil logn \rceil$ many triples, each of length at most $3logn$. Thus, the total space is $O(log^2 n)$

□

**Theorem 53.** *$NSPACE(f(n)) \subseteq SPACE(f^2(n))$ for any proper complexity function $f(n)$.*

*Proof.* Let $Q \in NSAPCE(f(n))$ and let $\mathbb{M}$ be a NTM deciding $Q$, such that $\mathbb{M}$ is $f(n)$-space bounded. Let $x \in \{0, 1\}$ and consider $G(\mathbb{M}, x)$ (the configuration states on input $x \leq c^{f(n)}$). By the theorem of Savitch finding a path from a starting configuration on $x$ to a terminating configuration can be done by a deterministic TM in space

$$O(log^2 c^{f(n)}) = O(f^2(n))$$

. □

So in case of space complexity the gains of a nondeterministic TM compared to a deterministic TM are limited, in the sense that any problem a nondeterministic TM can solve, a deterministic can solve with the square of the required space by a non-determinstic TM and there are not exponential gains using later. In case of time boundaries this might not be the case as the exponential time hypothesis suggests. If the exponential time hypothesis should be true, than $\mathbb{P} \neq \mathbb{NP}$ would be true, see [15].

## 10. Conclusion

The two concepts presented in this thesis have a very different nature. On the one hand the theoretical bound to computation demonstrates a limit to what is effectively calculable. It stands on the solid and rigorous grounds of set theory itself. On the other hand the practical limitations of computation, governed by complexity issues is a way more intricate venture, because even the definition of different complexity classes into polynomial and exponential represent a certain kind of voluntary cutoff, because e.g there are algorithms in $\mathbb{EXP}$, that in all applications run faster, than algorithms in $\mathbb{P}$. For examples and a broader discussion of this see [3].

But even the strictness of theoretical limits lacks mathematical necessity in a very specific way, because Church's thesis, as convincing as it may be, is not a theorem one can prove [1], and there might be a way, however unlikely, to formalize the concept of effective calculability without running into a problem like the halting problem respectively undecidability. A strong argument in favor of Churs's thesis are of course Gödel's incompleteness results themselves, since the are equivalent to Turing's results about calculability [10], and it seems like all of mathematics, would have to be put on totally different ground to evade the problems Gödel's incompleteness theorems point out. But also in case of the practical limitation one can make a good argument for categorizing algorithms in these classes: On the one hand there are two paths of rejection against the discussed cutoff. First, for practical relevance like stated above, there are are algorithms that have a exponential running time but the exponent is rather small, e.g $e^{\frac{1}{e^{100}}}$ compared to algorithms, that run in polynomial time but the exponent is huge, e.g. $n^{1000}$. But [3] points out, that these are mostly exceptions and to built a solid theory it are the limiting cases, that define what is possible and which resemble a compass for mathematical north. Secondly it could be, that in the future it will be possible to create a computation, that is way more potent than deterministic TMs. It can be shown, that quantum TMs, with the ability to postselect, are capable of running algorithms of the complexity class $PP$, a class that contains $NP$,

see [13]. This could make the practical limitations of deterministic TMs irrelevant. Later of course only would be relevant on a theoretical level, if it turns out that $P \neq NP$.

It are the limiting cases, that are of interest to mathematicians, the realm of the possible, not the factual. (If there is a actual difference between them, would be a completely different question). The astonishing thing about the theoretical limits discussed in this thesis is not only why the exist, but that the exist at all. There is just no good reason to assume, that no matter how algorithms are formalized, the formalization produces equivalent classes of functions respectively all of the different approaches run into exactly the same kind of problem. Introducing infinity to the space of reason, seemingly enforces problems, that enable the fertile soil of mathematical ideas.

An interesting point of view would be to see infinity, in case of the theoretical limitation, as the source of the misery, since all problematic results only are relevant for infinite sets. But the question is if one should really view these problems a misery at all, since infinity is the clear thread, that is running through the vast field of mathematical research and makes mathematics so beautiful, by giving birth to seemingly paradox and contradicting results: Whether it is the axiom of choice, the continuum hypothesis, or in the case of this thesis the halting problem.

## References


[1] Enderton, Herbert B.: Computability Theory, An Introduction to Recursion Theory, University of Califonria, Los Angeles, 2011, p. 7-9

[2] Turing, Alan Mathison (Dec 1937). "Computability and Definability

[3] Papadimitriou, Christos M.: Computaional Complexity, Addison-Wesley Publishing Company, Inc., Boston, 1994

[4] Dickson, David (2000). "Mathematicians chase the seven million-dollar proofs". Nature. 405 (383): 383. doi:10.1038/35013216

[5] Rószar, Péter: Mathematische Annalen, Berlin 1935, Julius Springer Verlag

[6] Enderton, Herbert B.: An Open Mathematical Introduction to Logic

[7] Kleene SC. Introduction to Metamathematics. D. van Nostrand Co.; 1952. Ishi Press; 2009. https://doi.org/10.48550/arXiv.quant-ph/0412187

[8] Hilbert, David: Mathematische Annalen, Berlin 1926, Julius Springer Verlag, p. 189

[9] Aditya Ravishankar, Moksha Mevada and Ankit Kumar: Primitive recursive functions and computability Deparment of Computer Science and Automation,Indian Institute of Science, Bangalore, 2021

[10] Rosser Sr., John Barkley: "Informal Exposition of Proofs of Gödel's Theorem and Church's Theorem", Davis 1965

[11] Sipser, Michael: An introduction to computational complexity

[12] Sipser Michael (1978). "Halting Space-Bounded Computations". Proceedings of the 19th Annual Symposium on Foundations of Computer Science.

[13] Aranson, Scott: Quantum Computing, Postselection and Probabilistic Polynomial-Time,

[14] Savitch, Walter: Relationships between nondeterministic and deterministic tape complexities. In: Journal of Computer and System Sciences. Band 4, Nr. 2. Elsevier, 1970, ISSN 0022-0000, S. 177–192

[15] Impagliazzo, Russell; Paturi, Ramamohan (1999), "The Complexity of k-SAT", Proc. 14th IEEE Conf. on Computational Complexity,