Karlsruhe Institute of Technology

Complexity of the Sum of Square Roots Problem

Bachelor Thesis of

Jonathan Hunz

At the Department of Informatics
Institute of Theoretical Informatics

Reviewers:	PD Dr. Torsten Ueckerdt,
	Jun.-Prof. Dr. Thomas Bläsius
Advisors:	Miriam Goetze,
	Paul Jungeblut

Time Period: 22th November 2024 – 22th March 2024

Selbständigkeitserklärung

Ich versichere wahrheitsgemäß, diese Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Karlsruhe, 22. März 2024

Abstract

Comparing sums of square roots is inevitable in many geometric problems in Euclidean space, e.g. the Euclidean traveling salesman or Euclidean Steiner tree problem. The *Sum of Square Roots Problem* is a notorious open problem since the 1970s, and placing it in the vast landscape of complexity classes has been an ongoing challenge. Allender, Bürgisser, Kjeldgaard-Pedersen and Miltersen showed in 2009 that the Sum of Square Roots Problem lies within the Counting Hierarchy CH in their paper *On the Complexity of Numerical Analysis*. To date, no stronger upper bound for the complexity of this problem, let alone a placement in a well studied complexity class like P, NP or the Polynomial Hierarchy PH is known. In this work, we present the proof of Allender et al. and unpack its complicated intermediate steps.

Deutsche Zusammenfassung

Summen von Quadratwurzeln zu vergleichen ist in vielen geometrischen Problemen in euklidischen Räumen, wie zum Beispiel dem euklidischen Traveling Salesman Problem oder dem euklidischen Steiner-Baum-Problem, unvermeidlich. Seit den 1970er-Jahren ist das sogenannte *Sum of Square Roots Problem* eine bekannte offene Fragestellung und das Finden von Schranken an dessen Komplexität eine andauernde Aufgabe. Allender, Bürgisser, Kjeldgaard-Pedersen und Miltersen zeigen in *On the Complexity of Numerical Analysis*, erschienen 2009, dass das Sum of Square Roots Problem in der sogenannten Counting Hierarchie CH liegt. Seither ist keine stärkere obere Schranke an die Komplexität dieses Problems, geschweige denn eine Einordnung in eine der klassischen und gut erforschen Komplexitätsklassen P, NP oder die Polynomialzeithierarchie PH, bekannt. In dieser Arbeit präsentieren wir den Beweis von Allender et. al und erklären dessen vielschichtige Zwischenschritte.

Contents

1 Introduction **1**
- 1.1 Applications . . . 2
- 1.2 Related Work . . . 3
- 1.3 Outline . . . 4

2 Complexity Theory Basics **7**
- 2.1 Oracle Machines . . . 7
- 2.2 The Polynomial Hierarchy . . . 8
- 2.3 Counting Complexity . . . 9

3 Circuit Complexity **15**
- 3.1 Boolean Circuits . . . 15
- 3.2 Symmetric Functions . . . 17
- 3.3 NC, AC and TC . . . 17
- 3.4 Constant-Depth Circuits . . . 18
- 3.5 A Calculus for the Composition of Circuit Classes . . . 18
- 3.6 Uniformity . . . 19
- 3.7 Carry-Lookahead Addition . . . 20
- 3.8 Complexity of Addition . . . 21
- 3.9 Iterated Addition . . . 21

4 Number Theoretical Notions **25**
- 4.1 Chinese Remainder Representation . . . 25
- 4.2 Lagrange's Theorem . . . 28
- 4.3 Multiplicative Inverses . . . 28
- 4.4 The Multiplicative Group of the Integers modulo a Prime . . . 29
- 4.5 Binary Expansion of Prime Reciprocals . . . 31

5 Computation over the Reals **33**
- 5.1 Finite-Dimensional Machines . . . 33
- 5.2 The Blum-Shub-Smale Model . . . 35
- 5.3 Straight-Line Programs and the Problem PosSLP . . . 37
- 5.4 Boolean Parts . . . 37

6 PosSLP is in the Counting Hierarchy **39**
- 6.1 Dropping one Assumption . . . 45

7 SQRT-SUM is in the Counting Hierarchy **47**
- 7.1 Newton's Method . . . 47
- 7.2 Reducing SQRT-SUM to PosSLP . . . 48

Bibliography **51**

1. Introduction

The Pythagorean theorem is one of the most basic relations in geometry: Given the length of two sides a and b of a right triangle, the length of the hypotenuse c is the square root of the sum of a^2 and b^2. Distances between two points in Euclidean space can be computed by considering the difference of their Cartesian coordinates and then applying this fundamental statement. Moreover, if the goal is to decide whether the length of a path along a set of points is less than some value, comparing a sum of square roots to this value is inevitable. The task of deciding whether $\sum_{i=1}^{n} \sqrt{a_i} \geq k$ for nonnegative integers $a_1, \ldots, a_n$ and $k \in \mathbb{N}$ hence arises naturally in computational geometry. This problem or, to be precise, a slight generalization of it, is called the *Sum of Square Roots Problem* (SQRT-SUM) and is a notorious open problem since the 1970s. Multiple definitions are commonly used throughout literature. Out of the two presented below, we will stick to the first one throughout this text.

Problem 1.1. SQRT-SUM:

- Given nonnegative integers $a_1, \ldots, a_n$ and $\delta_i \in \{-1, 1\}$ for $i = 1, \ldots, n$, decide whether
$$\sum_{i=1}^{n} \delta_i \sqrt{a_i} \geq 0.$$
- Or equivalently, given integers $a_1, \ldots, a_n$ and $b_1, \ldots, b_m$, decide whether
$$\sum_{i=1}^{n} \sqrt{a_i} \geq \sum_{i=1}^{m} \sqrt{b_i}.$$

It is commonly agreed upon that Garey, Graham and Johnson were the first to mention SQRT-SUM in 1976 [GGJ76]. Since then, placing this problem in the vast landscape of complexity classes has been an ongoing challenge. Allender, Bürgisser, Kjeldgaard-Pedersen and Miltersen showed in 2009 that SQRT-SUM lies within the *Counting Hierarchy* CH in their paper *On the Complexity of Numerical Analysis* [All+09]. To date, no stronger upper bound on the complexity of SQRT-SUM, let alone a placement in a well studied complexity class like P, NP or the *Polynomial Hierarchy* PH is known.

In this work, we present the proof of Allender et al. and unpack many of its complicated intermediate steps. A great part of this proof concerns a problem called PosSLP that asks whether an integer represented by a simple sequence of arithmetic instructions is positive.

Theorem 1.2 ([All+09])**.** PosSLP *is in the counting hierarchy* CH*.*

Allender et al. show that this problem is in the third level of the counting hierarchy. A lemma used in this proof, however, contains two arguments that we are not able to validate. We present this auxiliary theorem in detail, highlight the two critical steps and prove it under the assumption that they are valid. Furthermore, we show a slightly weaker version which only depends on one of the assumptions and discuss the effects on the complexity of PosSLP.

Newton's method, an algorithm for approximating roots of rational functions, is used to compute the square root of any nonnegative integer up to desired precision. Using this algorithm, SQRT-SUM can be decided in polynomial time on a computational model called *Blum-Shub-Smale machine*, which is more suited for numerical computations than a standard Turing machine. We show how a certain kind of computations on such a machines can be simulated by a polynomial-time Turing machine given access to an *oracle* that can decide instances of PosSLP in constant time. Theorem 1.2 is then used to prove the main theorem of this text:

Theorem 1.3. *SQRT-SUM is in counting hierarchy* CH*.*

Throughout this text, we assume only basic mathematical and complexity theoretical knowledge. For this reason, a great part of it introduces new concepts that are needed to understand the complex result. We try to explain all steps in a comprehensible manner; the proofs of some statements, however, are out of the scope of this work.

1.1 Applications

SQRT-SUM arises in many geometric problems. The *Euclidean Traveling Salesman Problem* and the *Euclidean Steiner-Tree Problem* are two prominent examples:

Problem 1.4. Euclidean-TSP: Given a set $N \subseteq \mathbb{Z}^2$ of points in the Euclidean plane, find a *tour* of minimal length, where the distance between two points is defined using the Euclidean norm.

Problem 1.5 ([Bra+14], p. 329)**.** Euclidean-STEINER-TREE: Given a set $N \subseteq \mathbb{Z}^2$ of points in the Euclidean plane, find a tree $T = (V, E)$ embedded in $\mathbb{Z}^2$, such that $N \subseteq V$ and $\sum_{(u,v)\in E} \|u - v\|$ is minimal, where $\|\cdot\|$ is the Euclidean norm.

Both problems can be generalized to $\mathbb{Z}^d$ for some fixed $d \in \mathbb{N}$. Furthermore, each of them has associated with it a decision problem asking whether there exists a tour or, respectively, Steiner tree having at most some given length. Not only does the task of comparing sums of square roots appear in these problems, their complexity hinges on SQRT-SUM. Neither Euclidean-TSP nor Euclidean-STEINER-TREE is known to be in NP. For the former, however, we show that this is indeed the case if there exists a polynomial-time algorithm for SQRT-SUM.

Theorem 1.6. SQRT-SUM $\in$ P $\implies$ Euclidean-TSP $\in$ NP

Proof. An instance of the decision version of Euclidean-TSP consists of a set of k points in $\mathbb{Z}^d$ and a length L. We have to show that for a certificate, i.e. a tour, it can be verified in polynomial time that the total length is less than L. The length of a tour visiting the

points $v_1, \ldots, v_k$ in order is given by the sum of their relative distances. The Euclidean distance between two point $u, v \in \mathbb{Z}^d$ is defined as

$$d(u, v) = \|u - v\| = \sqrt{\sum_{i=1}^{d} (u_i - v_i)^2}.$$

This expression is the square root of a nonnegative integer, therefore the total length of the tour $v_1, \ldots, v_k$ is a sum of square roots. If SQRT-SUM $\in$ P, it can be decided in polynomial time by a deterministic TM whether this sum is less than L. Hence, Euclidean-TSP $\in$ NP if SQRT-SUM $\in$ P. □

As a side note, we want to mention some interesting results concerning the two presented problems: Garey et al. showed that the problem EXACT-COVER-BY-3-SETS, which is NP-complete, can be reduced to Euclidean-STEINER-TREE and Euclidean-TSP when the rectilinear (Manhattan) metric or a discretized version of the Euclidean metric is used instead of the Euclidean metric [GGJ76]. Furthermore, these versions of the two problems are in NP, hence they are NP-complete. Independently, Papadimitriou showed that a variant of Euclidean-TSP where the Euclidean metric is used but distances are rounded to integers is also NP-complete [Pap77].

Aside from that, Arora and, independently, Mitchell proved the existence of *polynomial-time approximation schemes* (PTAS) for Euclidean-TSP and Euclidean-STEINER-TREE [Aro98], [Mit99]. For a minimization problem, a PTAS is a family of algorithms that for a fixed $\varepsilon > 0$ compute, in time polynomial in the size of the problem instance, a solution that is within a factor of $(1 + \varepsilon)$ of the optimal solution. E.g. if for a given instance of Euclidean-TSP the optimal solution has a total length of L, a PTAS would compute a solution that has a length of at most $(1 + \varepsilon) \cdot L$. Arora and Mitchell were awarded the 2010 Gödel Prize for this results.

1.2 Related Work

1.2.1 Complexity Results

Deciding whether a sum of square roots is equal to zero can be solved by a deterministic polynomial-time algorithm, as proven by Blömer [Blö91]. However, if the sum is not equal to zero, this does not solve the problem of determining its sign.

SQRT-SUM is in the complexity class $\exists\mathbb{R}$ which consists of all problems that can be reduced to the *existential theory of the reals*. Since $\exists\mathbb{R} \subseteq$ PSPACE [Can88, Theorem 3.3], SQRT-SUM can be decided in PSPACE. It is unlikely that SQRT-SUM is also PSPACE-complete. We provide more evidence for this in Section 2.3.2.

Malajovich stated a conjecture concerning *Kronecker's Theorem*, a result from the mathematical branch of Diophantine approximation, relative to which SQRT-SUM $\in$ P [Mal01]. Theorem 1.6 states that if this conjecture is true, Euclidean-TSP is in NP.

Etessami and Yannakakis showed that SQRT-SUM reduces to the *Qualitative Termination Problem for Recursive Markov Chains* [EY09] and the problem of computing a *Nash Equilibrium* of a specific game [EY10].

In November 2023, Balaji and Datta showed that a unary version of SQRT-SUM is in the complexity class P/poly [BD23]. This class is the set of decision problems that can be decided by *small circuits* or equivalently Turing machines that are given a polynomial length *advice string*.

1.2.2 Separation Bounds

A large area of work concerning the sum of square roots problem is the study of *separation bounds*. For an arithmetic expression E having a value ξ, a separation bound $\text{sep}(E)$ is a positive real number such that if $\xi \neq 0$, then $|\xi| \geq \text{sep}(E)$. In 1981, O'Roukre proposed the problem of finding the minimal positive value of the expression $\sum_i \sqrt{a_i} - \sum_j \sqrt{b_j}$, where a_i and b_j are positive integers with $1 \leq a_i, b_j \leq N$ for some N and $i, j \in \{1, 2\}$ [ORo81]. Angluin and Eisenstat showed that for any two nonnegative integers a and b between 1 and N, the minimum nonzero distance of $\sqrt{a} + \sqrt{b}$ to an integer is in $\Theta(1/N^{3/2})$ [AE04]. Burnikel et al. proved the existence of an easily computable separation bound for expressions involving radicals, i.e. square roots, in [Bur+00]. As stated in [EHS23], the separation bound presented in [Bur+00] implies that

$$\left| \sum_{i=1}^{n} \delta_i \sqrt{a_i} \right| \geq \left(n \max_{i=1,\ldots,n} \sqrt{a_i} \right)^{-(2^n - 1)}.$$

In their 2023 paper [EHS23], Eisenbrand et al. provided a new bound of

$$\left| \sum_{i=1}^{n} \delta_i \sqrt{a_i} \right| \geq \gamma n^{-2n},$$

where γ is a constant that depends on the numbers $a_1, \ldots, a_n$. Without diving any deeper into the topic of separation bounds, the following example might give a sense for why sums of square roots are hard to compare.

Example ([DMO09])**.** For $n, k \in \mathbb{N}$, let $\text{r}(n, k)$ be the minimum nonzero value of the expression

$$\left| \sum_{i=1}^{n} \sqrt{a_i} - \sum_{i=1}^{m} \sqrt{b_i} \right|,$$

where $a_1, \ldots, a_k$ and $b_1, \ldots, b_k$ are integers with $0 \leq a_i, b_i \leq n$ for $i \in \{1, \ldots, k\}$. The minimum nonzero distance between the sum of square roots of two and, respectively, three integers less than or equal to 20 is given by

$$\text{r}(20, 2) = \sqrt{10} + \sqrt{11} - \sqrt{5} - \sqrt{18} \approx 1.9 \cdot 10^{-4} \quad \text{and}$$

$$\text{r}(20, 3) = \sqrt{5} + \sqrt{6} + \sqrt{18} - \sqrt{4} - \sqrt{12} - \sqrt{12} \approx 4.8 \cdot 10^{-6}.$$

We see that by allowing one more summand, the minimum nonzero distance decreased by two orders of magnitude.

1.3 Outline

In Chapter 2 we introduce important concepts from complexity theory: Oracle Turing machines allow us to characterise the complexity of decision problems under the assumption that we can treat certain parts of an algorithm as a black box. We generalize the concept of language oracles to oracles for whole complexity classes and show basic relations concerning them. The *polynomial hierarchy* PH is introduced, which is a generalization of the classes NP and coNP. We explain the concept of counting complexity and the complexity classes $\#\mathsf{P}$ and PP. Toda's theorem, an important result which states that every problem in the polynomial hierarchy can be reduced to counting, is presented. At the end of Chapter 2, we define the *counting hierarchy*.

Chapter 3 gives a detailed definition of Boolean circuits and introduces complexity classes that capture their computational power and limitations. A calculus for the composition of

different circuit classes is presented. Boolean circuits are, in general, a *non-uniform* model of computation. We discuss the rather absurd consequences of this property and how this model can be fixed. The rest of Chapter 3 considers circuits for addition and their complexity: The *carry-lookahead* method for addition is presented as a more efficient alternative for the classical *carry-ripple* method. The problems ADD, ITADD and LOGITADD are introduced and placed within the framework of composed circuit complexity classes. Lastly, we show how these addition circuits can be simulated on a Turing machine, in order to prove upper bounds on the complexity of addition in terms of the complexity classes defined in Chapter 3.

The proof of Theorem 1.2 (PosSLP $\in$ CH) makes use of statements from number theory, which we cover in Chapter 4. First and foremost, we introduce the *Chinese remainder representation*, which allows us work with "very large" numbers by regarding them as a list of remainders modulo some primes. Next, we recall basic definition from group theory and show some fundamental theorems. We explore the *multiplicative group of the integers modulo a prime* in more detail and prove that is *cyclic*. Finally, we briefly explain how to easily compute bits of the binary expansion of prime reciprocals. Most of the statements presented in this chapter are not hard to prove and can be found in introductory literature on number theory or algebra. In our opinion, however, it is helpful to have a sense for *why* they are true in order to fully understand how they are used as arguments.

Chapter 5 contains an extensive description of a computational model called the *Blum-Shub-Smale machine*, which is a generalization of the standard Turing machine. This machine operates on a tape consisting of elements of an arbitrary ring or field and can perform arithmetic operations on its cells in constant time. *Straight-line programs*, which are sequences of arithmetic instructions, are introduced as another model of arithmetical computation. We give an example on how these simple programs can represent large numbers, relative to their size. We define the problem PosSLP, which asks whether a straight-line program evaluates to a positive number. The concept of *Boolean parts* forms the link between machines over the real numbers and standard Turing machine operating on Boolean values. Using Boolean parts and the problem PosSLP, we then relate the world of *real computation* to the classical Turing framework.

In Chapter 6, we use the preliminaries established in the preceding chapters to prove that PosSLP lies within the counting hierarchy. We split the proof of Allender et al. into small parts. As already mentioned, the proof contains two arguments, which we are not able to validate. We highlight these arguments and show the concerning lemma under the assumption that they are true. In addition, we show a slightly weaker statement which depends only on one of the assumptions.

The square root of any nonnegative integer can be approximated using an algorithm known as *Newton's method*, which we describe in Chapter 7. Together with the preliminaries established in the previous chapters and the result about PosSLP, we then prove our main theorem, which states that SQRT-SUM is in the counting hierarchy.

2. Complexity Theory Basics

In this chapter, we introduce important concepts from complexity theory that are essential for the rest of this text. We assume familiarity with terms such as deterministic and nondeterministic Turing machines, decidability, and the classes P, NP and coNP. To begin, let us briefly recall the concept of a *polynomial-time Karp reduction*:

Definition 2.1 ([AB07], Definition 2.7)**.** A language $L \in \{0,1\}^*$ is *polynomial-time Karp reducible* to a language $L' \in \{0,1\}^*$, if there is a polynomial-time computable function $f\colon \{0,1\}^* \to \{0,1\}^*$, called the *reduction*, such that for every $x \in \{0,1\}^*$, $x \in L$ if and only if $f(x) \in L'$.

2.1 Oracle Machines

Definition 2.2 ([AB07], Definition 3.4)**.** An *oracle Turing machine* M^O, where $O \subseteq \{0,1\}^*$ is called the *oracle*, is a standard (deterministic or non-deterministic) Turing machine (TM) that has an additional read-write tape called the *oracle tape* and states q_{query}, q_{yes}, and q_{no}. Whenever M enters the state q_{query} and the content of the oracle tape is q, M moves into q_{yes} if $q \in O$ and q_{no} if $q \notin O$ in the next computation step.

Definition 2.3 ([AB07], Definition 3.5)**.** For every $O \subseteq \{0,1\}^*$, P^O and NP^O denote the sets of languages that can be decided by a polynomial-time deterministic TM and polynomial-time non-deterministic TM, respectively, having oracle access to O.

Let us briefly recall the definition of a NP-hard and NP-complete language: A language L is NP-hard if every $L' \in \mathsf{NP}$ can be polynomially-time Karp reduced to L. If L is also a member of NP, we say that L is NP-complete. [AB07, Definition 2.7]

We can generalize the notions of hardness and completeness for arbitrary complexity classes: Let $\mathcal{L}$ be some set of languages, then a language L is *$\mathcal{L}$-hard* if every $L' \in \mathcal{L}$ can be reduced to L and *$\mathcal{L}$-complete* if additionally $L \in \mathcal{L}$. Note that in this case *reduced* does not necessarily mean that the reduction is a polynomial-time Karp reduction.

The language SAT consists of all satisfiable Boolean formulas in conjunctive normal form (CNF). By the above definition, P^{SAT} is the set of all languages that can be decided by a polynomial-time TM M having access to a SAT-oracle. That is, M can write any (polynomial-length) CNF-formula F on its oracle tape and after entering the query state, M

transitions to either the state q_{yes} or q_{no} in the next step, depending on whether $F \in \text{SAT}$. Since SAT is NP-complete, M can actually use the oracle to answer queries about the membership of some polynomial-length string $w \in \{0,1\}^*$ in any language in NP. This can be accomplished by first transforming w into a polynomial-sized CNF-formula F_w, which is possible since there exists a polynomial-time Karp reduction, and then sending F_w to the SAT-oracle. [AB07, p. 45]

Definition 2.4. Let $\mathcal{L}$ be a set of languages, then $\mathsf{P}^{\mathcal{L}} = \bigcup_{L \in \mathcal{L}} \mathsf{P}^L$. Furthermore, if there exists an $\mathcal{L}$-complete language L in the sense of polynomial-time Karp reducibility, then $\mathsf{P}^{\mathcal{L}} = \mathsf{P}^L$.

Lemma 2.5. $\mathsf{P}^{\mathsf{P}} = \mathsf{P}$

Proof. $\mathsf{P} \subseteq \mathsf{P}^{\mathsf{P}}$ since a TM with access to a P-oracle can simply not use this capability. For the other direction let L be a language in P^{P}. By definition, there exists a language $L' \in \mathsf{P}$, such that $L \in \mathsf{P}^{L'}$, i.e. L can be decided by a polynomial-time TM M having access to a L'-oracle. Since $L' \in \mathsf{P}$, L' is also decided by a polynomial-time TM. Every query of the L'-oracle can thus be simulated in polynomial-time by M and therefore $L \in \mathsf{P}$. □

Remark. Definition 2.4 states that if there exists a language L' that is complete for some set of languages $\mathcal{L}$, then a $\mathcal{L}$-oracle can be replaced by a L'-oracle. This is under the assumption that the reduction can actually be performed by the machine using the oracle. Not all complexity classes are known to have complete languages, however, this is the case for the class P. A language L' is P-complete it is in P and every language in P is *log-space* reducible to it [AB07, Definition 6.28]. A machine using $\mathcal{O}(\log n)$ space can only need a polynomial amount of time, since there exists at most $2^{\mathcal{O}(\log n)}$ possible configurations. For this reason, log-space reducibility implies polynomial-time Karp reducibility. An example of a P-complete language is the language CIRCUIT-EVAL, which we do not discuss in further detail [AB07, Theorem 6.30]. We might therefore show the direction $\mathsf{P}^{\mathsf{P}} \subseteq \mathsf{P}$ in the proof of Lemma 2.5 using the concept of P-completeness: Let L be language in P^{P}, thus L is decided by a polynomial-time TM that has access to an oracle for a P-complete language L'. A polynomial-time TM M can simulate this oracle by transforming a query string w into an instance w' of L', since the P-completeness of L' implies that every language in P is polynomial-time Karp reducible to L'. Afterwards, M decides whether w' is a member of L' in polynomial time.

2.2 The Polynomial Hierarchy

Definition 2.6 ([AB07], Definition 5.3)**.** The *polynomial hierarchy* is the set $\mathsf{PH} = \bigcup_{i \in \mathbb{N}_0} \Sigma_i^p$, where Σ_i^p is the set of languages for which there exists a polynomial-time TM M and a polynomial q such that

$$x \in L \iff Q_1 u_1 \in \{0,1\}^{q(|x|)} \dots Q_i u_i \in \{0,1\}^{q(|x|)} M(x, u_1, \dots, u_i) = 1$$

with Q_i denoting $\forall$ or $\exists$ depending on whether i is even or odd, respectively.

Remark. Alternatively, let Π_i^p be defined analogously to Σ_i^p but the existential and universal quantifiers are swapped, i.e. Q_i is $\forall$ is i is odd and $\exists$ if i is even. Then $\mathsf{PH} = \bigcup_{i \in \mathbb{N}_0} \Pi_i^p$.

We give some interesting facts about the polynomial hierarchy without proof: The well known complexity classes P, NP and coNP form the foundation of PH in the sense that $\mathsf{P} = \Sigma_0^p = \Pi_0^p$, $\Sigma_1^p = \mathsf{NP}$ and $\Pi_0^p = \mathsf{coNP}$. For every $i \in \mathbb{N}_0$, the inclusions $\Sigma_i^p \subseteq \Pi_{i+1}^p \subseteq \Sigma_{i+2}^p$

hold. If Σ_i^p is strictly contained in Σ_{i+1}^p for every $i \in \mathbb{N}_0$ then, in particular, $\mathsf{P} \neq \mathsf{NP}$. In this case we say that the polynomial hierarchy *does not collapse*. On the other hand, if there exists some $i \in \mathbb{N}_0$ such that $\Sigma_i^p = \Sigma_{i+1}^p$, PH *collapses to the i-th level*, that is $\mathsf{PH} = \Sigma_i^p$. The polynomial hierarchy collapses to the first level, i.e. $\mathsf{PH} = \mathsf{P}$, if and only if $\mathsf{P} = \mathsf{NP}$. [AB07, p. 97]

Similar to Definition 2.4 we define PH^O to be the set of languages as defined in Definition 2.6 but the TM M has access to an oracle O for some language (or complexity class).

Definition 2.7. Let $\mathcal{L}$ be a set of languages, then $\mathsf{PH}^{\mathcal{L}} = \bigcup_{L \in \mathcal{L}} \mathsf{PH}^L$. Furthermore, if there exists an $\mathcal{L}$-complete language L in the sense of polynomial-time Karp reducibility, then $\mathsf{PH}^{\mathcal{L}} = \mathsf{PH}^L$.

In continuation of Lemma 2.5 we show two more equivalences concerning oracles that we use frequently in this text:

Lemma 2.8. $\mathsf{PH}^{\mathsf{PH}} = \mathsf{PH}$

Proof. Trivially, $\mathsf{PH} \subseteq \mathsf{PH}^{\mathsf{PH}}$. To show the other inclusion let L be a language in $\mathsf{PH}^{\mathsf{PH}}$. By Definition 2.7, there exists a language $L' \in \mathsf{PH}$, such that $L \in \mathsf{PH}^{L'}$. Hence, there exists a polynomial-time TM M having access to a L'-oracle, alternating quantifiers $Q_1, \ldots, Q_i$, and a polynomial p, such that

$$x \in L \iff Q_1 u_1 \in \{0,1\}^{p(|x|)}, \ldots, Q_i u_i \in \{0,1\}^{p(|x|)} \colon M(x, u_1, \ldots, u_i) = 1.$$

Also, since $L' \in PH$, there exists a polynomial-time TM M', alternating quantifiers $Q'_1, \ldots, Q'_j$, and a polynomial p', such that

$$x \in L' \iff Q'_1 u'_1 \in \{0,1\}^{p'(|x|)}, \ldots, Q'_j u'_j \in \{0,1\}^{p'(|x|)} \colon M'(x, u'_1, \ldots, u'_j) = 1.$$

We can construct a TM $\widetilde{M}$ that simulates M when given a tuple $(x, u_1, \ldots, u_i, u'_1, \ldots, u'_j)$. Every time that M asks its L'-oracle about the membership of some string w, $\widetilde{M}$ runs $M'(w, u'_1, \ldots, u'_j)$ and transitions into the appropriate state depending on the answer. Hence, there exists a polynomial $\widetilde{p}$ such that

$$x \in L \iff Q_1 u_1 \in B, \ldots, Q_i u_i \in B, Q'_1 u'_1 \in B, \ldots, Q'_j u'_j \in B \colon \widetilde{M}(x, u_1, \ldots, u_i, u'_1, \ldots u'_j),$$

where $B = \{0,1\}^{\widetilde{p}(|x|)}$. If $Q_i = Q'_1$, i.e. the chain of quantifiers is not alternating, we can merge the two quantifiers into a new expression quantifying over strings of length $2\widetilde{p}(|x|)$ and split the certificates into u_i and u'_1 afterwards. Thus $L \in \mathsf{PH}$. □

Corollary 2.9. $\mathsf{PH}^{\mathsf{P}} = \mathsf{PH}$

Proof. The claim follows immediately from Lemma 2.8 since $\mathsf{P} \subseteq \mathsf{PH}$. □

2.3 Counting Complexity

The complexity classes P and NP capture the difficulty of finding and validating certificates, respectively. It is an open question whether these tasks are equivalently difficult. The polynomial hierarchy presented in the previous section generalizes these fundamental complexity classes by allowing the quantification of certificates. We now introduce the concept of *counting complexity* that concerns with determining the total number of certificates. Using the language SAT as an example, we demonstrate how decision problems in NP have corresponding counting problems. These counting problems in turn can be transformed back into decision problems.

Definition 2.10 ([AB07], Definition 17.5)**.** #P is the set of functions $f\colon \{0,1\}^* \to \mathbb{N}_0$ for which there exists a polynomial p and a polynomial-time TM M such that for every $x \in \{0,1\}^*$

$$f(x) = \left|\left\{y \in \{0,1\}^{p(|x|)} : M(x,y) = 1\right\}\right|.$$

The class #P, pronounced "sharp P", can also be defined in terms of nondeterministic TMs. That is, a function f is in #P if, for every input $x \in \{0,1\}^*$, the value $f(x)$ is equal to the number of accepting paths of a nondeterministic polynomial-time TM.

Example. #SAT is the corresponding counting problem to the NP-complete problem SAT. In case of #SAT, the task is to compute the total number of satisfying assignments for a Boolean formula. A Boolean formula is satisfiable if and only it has at least one satisfying assignment. Solving an instance of #SAT must therefore, in some sense, be at least as hard as deciding an instance of SAT. The function #SAT that maps a Boolean formula F to the number of satisfying assignments of F is in #P.

Definition 2.11 ([AB07], Definition 17.6)**.** PP is the set of languages for which there exists a polynomial-time TM M and a polynomial p such that for every $x \in \{0,1\}^*$

$$x \in L \iff \left|\left\{y \in \{0,1\}^{p(|x|)} : M(x,y) = 1\right\}\right| \geq \frac{1}{2} \cdot 2^{p(|x|)}.$$

Intuitively speaking, the concern of problems in PP is deciding the most significant bit of functions in #P. As a side note, the counterpart of PP is the complexity class $\oplus$P, pronounced "parity P". Problems in this class concern with determining the *least* significant bit of a function in #P [AB07, Definition 17.15]. As for #P, we can also define PP in terms of nondeterministic TMs: A language L is in PP if there exists a nondeterministic TM M such that $x \in L$ if and only if the majority of computation paths of M are accepting, for every $x \in \{0,1\}^*$.

Example. MAJ-SAT is a decision problem that asks whether the majority of assignments for a given Boolean formula is satisfying. For example, let F be a Boolean formula of n variables, then F is a *yes*-instance of MAJ-SAT, if and only if at least 2^{n-1} of the possible 2^n possible assignments are satisfying. The language MAJ-SAT is in the complexity class PP. Let us summarize the relationship between the classes NP, #P and PP:

Decision Problem	**Counting Problem**	**Decision Problem**
SAT $\in$ NP	#SAT $\in$ #P	MAJ-SAT $\in$ PP
Is there a satisfying assignment?	*How many satisfying assignments are there?*	*Is the majority of assignments satisfying?*

Remark. A *probabilistic* Turing machine is a TM that may take possibly multiple transitions at each step and chooses which transition to use according to some probability distribution. Although probabilistic TMs are similar to nondeterministic TMs, they differ in how we interpret their acceptance behaviour. We say that a nondeterministic TM accepts an input if there exists at least one accepting path. However, for a probabilistic Turing machine we consider the probability that a path is accepting. If the underlying probability distribution is uniform, this is equal to the fraction of all accepting paths. Instead of a machine that chooses each transitions at random, one could also use a deterministic TM and provide a series of "coin tosses" that determine the transitions as additional input. [AB07, p. 125]

The class PP can also be defined in this probabilistic framework: Let us denote the probability of an event $X(r)$, where r is sampled from a (uniform) probability distribution R, as $\Pr_{r \leftarrow R}[X(r)]$. Furthermore, for a language L, we define the function $L\colon \{0,1\}^* \to \{0,1\}$ to be 1 if $x \in L$ and 0 otherwise for every $x \in \{0,1\}^*$. A language L is in PP if and only if there exists a deterministic polynomial-time TM and a polynomial p such that

$$\Pr_{r \leftarrow \{0,1\}^{p(|x|)}}[M(x,r) = L(x)] > \frac{1}{2}$$

for every input $x \in \{0,1\}^*$. Here, we provide M with a polynomial-length series of "coins tosses" and thereby simulate a probabilistic TM, as stated above.

The related complexity class BPP, which stands for *bounded error probabilistic polynomial time*, is commonly defined as the set of languages L for which exists a polynomial-time *probabilistic* TM M and a polynomial p, such that

$$\Pr_{r \leftarrow \{0,1\}^{p(|x|)}}[M(x,r) = L(x)] \geq \frac{2}{3}$$

for every input $x \in \{0,1\}^*$. In fact, the constant $2/3$ for BPP and $1/2$ in the case of PP can be replaced by $1/2 + \varepsilon$ for some constant $\varepsilon > 0$ and $1/2 + 2^{-|x|}$, respectively. This brings us to the key difference between these two complexity classes: For any decision problem in BPP, we can increase the probability that the output of the TM (accept/reject) is actually correct by running the machine polynomially many times, the emphasis being on polynomially. This process is called *probability amplification*. For a problem in PP, in general, it would take exponentially many trials to distinguish between the two cases. [AB07, p. 345]

It is evident, that we can decide whether the majority of assignments of a Boolean formula is satisfying, if the total number of satisfying assignments is known. We now show that given access to an oracle which decides if the majority of assignments is satisfying, determining the total number of satisfying assignments can be accomplished in polynomial time by performing a binary search.

Lemma 2.12 ([Fil15], [For02])**.** $\mathsf{P}^{\#\mathsf{P}} = \mathsf{P}^{\mathsf{PP}}$

Proof. To show $\mathsf{P}^{\mathsf{PP}} \subseteq \mathsf{P}^{\#\mathsf{P}}$ let M be a deterministic polynomial-time TM having access to a PP-oracle. That is, M has access to an oracle which can answer queries about the membership of an arbitrary $x \in \{0,1\}^*$ in a PP-complete language and thus any language in PP. We have to show that we can simulate such an oracle using a #P-oracle. Let L be a language in PP. By Definition 2.11, there exists a polynomial p, such that a $x \in \{0,1\}^*$ is a member of L if and only if the majority of strings $y \in \{0,1\}^{p(|x|)}$ are certificates. Using the #P-oracle we can find total number of certificates of length $p(|x|)$ and can decide in polynomial time if this number is at least $1/2 \cdot 2^{p(|x|)}$.

The direction $\mathsf{P}^{\#\mathsf{P}} \subseteq \mathsf{P}^{\mathsf{PP}}$ requires more work: Let L be a language in $\mathsf{P}^{\#\mathsf{P}}$. L can be decided by a polynomial-time TM having access to an oracle for a function $f \in \#\mathsf{P}$, i.e. $L \in \mathsf{P}^f$. We show that a call to the f-oracle call can be simulated in polynomial time given access to a PP-oracle.

By the definition of #P, there exists a nondeterministic polynomial-time TM M such that for every input $x \in \{0,1\}^*$, the value $f(x)$ is equal to the number of accepting paths of M. For all inputs $x \in \{0,1\}^*$, denote the total number of computation paths of M as $T^M(x)$ and let $T_1^M(x)$ and $T_0^M(x)$ denote number of accepting and rejecting paths of M, respectively. Furthermore, we define the following two languages:

$$L_1 = \Big\{(x,y) \in \{0,1\}^* \times \mathbb{N}_0 \mid T^M(x) \geq y\Big\} \quad \text{and}$$
$$L_2 = \Big\{(x,y) \in \{0,1\}^* \times \mathbb{N}_0 \mid T_1^M(x) \geq y\Big\}.$$

We show the following four statements:

1. $L_1 \in \mathsf{PP}$.
2. Given a L_1-oracle, $T^M(x)$ is computable in polynomial time using binary search.
3. $L_2 \in \mathsf{PP}$.
4. Given a L_2-oracle, $T_1^M(x)$ is computable in polynomial time using binary search.

The following proof of these statements yields a polynomial-time algorithm for computing $T_1(x)$, assuming we are given access to a PP-oracle.

1. We construct a nondeterministic TM M' and show that, on input x, the majority of computation paths of M' is accepting if and only if M has at least y accepting paths, i.e.
$$T_1^{M'}(x) \geq \frac{1}{2}T^{M'}(x) \iff T^M(x) \geq y.$$
Since the language L_1 is exactly the set of tuples (x, y) such that $T^M(x) \geq (y)$, the above statement implies that $L_1 \in \mathsf{PP}$.

   M' starts by nondeterministically choosing a bit $b \in \{0, 1\}$. Note that in the probabilistic model, we would additionally require that b is chosen from the *uniform* distribution $\{0, 1\}$. If $b = 1$, M' nondeterministically chooses a value $z \in \{1, 2, 3\}$. If $z = 1$, M' accepts immediately. Otherwise, that is if $z = 2$ or $z = 3$, M' simulates M on the input x and accepts afterwards. This procedure generates $2T^M(x) + 1$ computational paths each of which is accepting.

   If $b = 0$, M' generates $2y$ paths, each of which rejects the input. This can be accomplished by, for example, nondeterministically writing a word in $\{1^k \mid k \leq 2y\}$ onto the tape and rejecting afterwards.

   The total number of computation paths of M' is the sum of accepting and rejecting paths, i.e.
$$T^{M'}(x) = T_0^{M'}(x) + T_1^{M'}(x) = 2y + 2T^M(x) + 1.$$
   Therefore, we have the following equivalences:
$$\begin{aligned} & T_1^{M'}(x) \geq \frac{1}{2}T^{M'}(x) \\ \iff \quad & 2T^M(x) + 1 \geq \frac{1}{2}(2y + 2T^M(x) + 1) \\ \iff \quad & T^M(x) + \frac{1}{2} \geq y \\ \iff \quad & T^M(x) \geq y, \end{aligned}$$
   where the last equivalence follows because $T^M(x) \in \mathbb{N}$.

2. M is a nondeterministic polynomial-time TM. Hence, there exists a polynomial p such that M performs at most $p(|x|)$ steps on input x. Without loss of generality, we can assume that M nondeterministically chooses between two transitions in each step. For every input x, the number of computational paths its therefore at most $2^{p(|x|)}$. Given access to a L_1-oracle, we perform a binary search using an initial lower and upper bound of 0 and $2^{p(|x|)}$, respectively. A binary search on a set of size $2^{p(|x|)}$ takes at most $\log(2^{p(|x|)}) = p(|x|)$ steps. Thus, $T^M(x)$ is be computable in polynomial-time given access to a L_1-oracle.

3. We construct a TM M'', such that

$$T_1^{M''}(x) \geq \frac{1}{2}T^{M''}(x) \iff T_1^M \geq y.$$

Like M', M'' begins by choosing a bit $b \in \{0,1\}$. If $b = 1$, M'' simulates M on the input x and accepts if and only if M accepts. Otherwise, that is if $b = 0$, M'' generates $T^M(x)$ computation paths; y rejecting and $T^M(x) - y$ accepting. By this construction, it follows that

$$T^{M''}(x) = 2T^M(x) \quad \text{and} \quad T_1^{M''}(x) = T_1^M(x) + T^M(x) - y.$$

Therefore

$$T_1^{M''}(x) \geq \frac{1}{2}T^{M''}(x) \iff T_1^M(x) + T^M(x) - y \geq T^M(x) \iff T_1^M(x) \geq y.$$

Again, this implies $L_2 \in \mathsf{PP}$.

4. Analogously to the second step, there exists a polynomial p such that the number $T_1^M(x)$ is bound by $2^{p(|x|)}$ from above. We perform binary search and obtain $T_1^M(x)$ in $\log(2^{p(|x|)}) = p(|x|)$ steps. Thus, $T_1^M(x)$ is computable in polynomial time given access to a L_2-oracle.

In summary, a TM having access to a PP-oracle, in order to query the languages L_1 and L_2, can compute $T_1^M(x)$ in polynomial time by performing two binary searches. Hence, we can simulate a $\#\mathsf{P}$-oracle in P^{PP}, which implies the claim $\mathsf{P}^{\#\mathsf{P}} \subseteq \mathsf{P}^{\mathsf{PP}}$. □

2.3.1 Toda's Theorem

In his seminal paper "PP is as Hard as the Polynomial-Time Hierarchy" ([Tod91]), Toda showed that the polynomial hierarchy is contained within P^{PP}. Toda was awarded the 1998 Gödel Prize for this result. The awarders justified their decision with the following words:

> "[...] Toda showed that two fundamental and much studied computational concepts had a deep and unexpected relationship. The first is that of alternation of quantifiers - if one alternates existential and universal quantifiers for polynomial time recognizable functions one obtains the polynomial time hierarchy. The second concept is that of counting the exact number of solutions to a problem where a candidate solution is polynomial time recognizable. Toda's astonishing result is that the latter notion subsumes the former - for any problem in the polynomial hierarchy there is a deterministic polynomial time reduction to counting. This discovery is one of the most striking and tantalizing results in complexity theory. It continues to serve as an inspiration to those seeking to understand more fully the relationships among the fundamental concepts in computer science." [Par99]

We now present Toda's theorem, another result from his work, and a useful Corollary that we use repeatedly in the subsequent chapters to connect statements about the polynomial hierarchy with those concerning counting complexity.

Theorem 2.13 (Toda's Theorem – [Tod91], Main Theorem)**.** $\mathsf{PH} \subseteq \mathsf{P}^{\mathsf{PP}}$

Theorem 2.14 ([Tod91], Theorem 4.10)**.** $\mathsf{PP}^{\mathsf{PH}} \subseteq \mathsf{P}^{\mathsf{PP}}$

Corollary 2.15. $\mathsf{PP}^{\mathsf{PH}^A} \subseteq \mathsf{P}^{\mathsf{PP}^A}$ *for every oracle* A*.*

2.3.2 The Counting Hierarchy

The *counting hierarchy* CH was introduced by Wagner in [Wag86] and independently by Parberry and Schnitger in [PS88].

It is similar to the previously defined polynomial hierarchy. However, CH builds a hierarchy over the class PP instead of NP.

Definition 2.16 ([AW97], Section 1)**.** Let $\mathsf{CH}_0^p = \mathsf{P}$, then for every $i \geq 0$

$$\mathsf{CH}_{i+1}^p = \mathsf{PP}^{\mathsf{CH}_i^p}.$$

The union $\bigcup_{i \in \mathbb{N}_0} \mathsf{CH}_i^p$ is denoted as CH. We refer to CH as the *counting hierarchy* and call CH_i^p its i-th level.

The counting hierarchy is contained in PSPACE and contains the polynomial hierarchy PH [AW97]. As stated in Chapter 1, SQRT-SUM $\in$ PSPACE and we show in Chapter 6 that SQRT-SUM $\in$ CH. Therefore, if SQRT-SUM is a PSPACE-complete problem, i.e. every problem in PSPACE can be polynomial-time reduced to SQRT-SUM, the counting hierarchy *collapses*.

3. Circuit Complexity

In this chapter, we present the computational model of *Boolean circuits*, together with complexity classes that capture their computational power and limitations. Boolean circuits are often studied out of the motivation that they are, in principle, easier to describe formally than Turing machines. Many researchers believe that it might therefore be easier to prove lower bounds on their complexity. Finding suitable circuit lower bounds could, in fact, prove that $\mathsf{P} \neq \mathsf{NP}$. Hence, this field of complexity theory has attracted great interest since the 1970s. However, large obstacles have prevented significant progress from being made in this area of work for many years. To phrase it in the words of Arora and Barak: Circuit lower bounds might be complexity theory's Waterloo. Another motivation to study the complexity of Boolean circuits comes from the fact that they resemble a simplified version of physical circuits used in actual computers. Results from circuit complexity could therefore answer, whether there exists tailor-made chips that are able to efficiently solve hard problems such as 3SAT. [AB07, p. 106]

Our goal though is to construct circuits for the efficient addition of binary numbers and prove upper bounds on the complexity of these problems. The complexity of addition will play a great role in the proof for Theorem 1.2. At the end of this chapter, we relate the world of circuit complexity, which deals with its own complexity classes, with the classical Turing framework by showing how these addition circuits can be simulated on Turing machines.

3.1 Boolean Circuits

Definition 3.1 ([Vol99], Definition 1.1, 1.2, 1.4)**.** A *Boolean function* is a function $f\colon \{0,1\}^n \to \{0,1\}$ for a $n \in \mathbb{N}$. A *family of Boolean functions* is a sequence $F = (f^n)_{n\in\mathbb{N}}$, where f^n is an n-ary Boolean function. A *basis* is a finite set of Boolean functions and families of Boolean functions.

Example. The Boolean functions $\wedge, \vee\colon \{0,1\}^2 \to \{0,1\}$ can easily be generalized to n-ary functions:

$$\begin{aligned} \mathrm{AND}^n\colon \{0,1\}^m \to \{0,1\}, &\quad (x_1,\ldots,x_n) \mapsto x_1 \wedge x_2 \ldots \wedge x_n \\ \mathrm{OR}^n\colon \{0,1\}^m \to \{0,1\}, &\quad (x_1,\ldots,x_n) \mapsto x_2 \vee x_2 \ldots \vee x_n. \end{aligned}$$

We can then define two families of Boolean functions:

$$\mathrm{AND} = (\mathrm{AND}^n)_{n\in\mathbb{N}} \quad \text{and} \quad \mathrm{OR} = (\mathrm{OR}^n)_{n\in\mathbb{N}}.$$

The basis $\{\text{AND}, \text{OR}, \neg\}$, where $\neg$ is the unary function mapping a Boolean value to its negation, is referred to as the *standard basis*.

We write $\text{F}(x_1, \ldots, x_n)$ instead of $\text{f}^n(x_1, \ldots, x_n)$ for a family of functions $F = (f^n)_{n \in \mathbb{N}}$ from now on. Note that a basis is always a finite set, but some elements may be families of functions and thus infinite objects themselves.

Definition 3.2 ([Vol99], Definition 1.6)**.** A *Boolean circuit* over a basis B with n inputs and m outputs is a tuple

$$C = (V, E, \alpha, \beta, \omega),$$

where (V, E) is a directed acyclic graph, $\alpha \colon E \to \mathbb{N}$ is an injective function, and functions $\beta \colon V \to B \cup \{x_1, \ldots, x_n\}$ and $\omega \colon V \to \{y_1, \ldots, y_n\} \cup \{*\}$, such that

1. If $v \in V$ has in-degree 0, then $\beta(v) \in \{x_1, \ldots, x_n\}$ or $\beta(v)$ is a Boolean constant.
2. If $v \in V$ has in-degree k with $k \geq 0$, then $\beta(v)$ is a k-ary Boolean function from B.
3. For every i, with $1 \leq i \leq n$, there is at most one node $v \in V$ such that $\beta(v) = x_i$.
4. For every i, with $1 \leq i \leq m$, there is at most one node $v \in V$ such that $\beta(v) = y_i$.

If a $v \in V$ has in-degree l and out-degree k we say that v has *fan-in* l and *fan-out* k. If $(u, v) \in E$, we say that u is a *predecessor gate* of v. If $\beta(v) = x_i$ for some i, then v is an *input node*, and if $\omega(v) = y_i$ for some i, then v is an *output node*.

The nodes $v \in V$ are also referred to as *gates*. Conceptually, the function β associates a gate with either the i-th bit of the input or a Boolean function from B. The function ω defines the output gates of the circuit, i.e. if $\omega(v) = y_i$, then v gives the i-th bit of the output. Since a Boolean functions $b \in B$ associated with a certain gate is in general not *symmetric* like the functions AND and OR, the value of b may depend on the order of its inputs. Thus, the function α is needed to define the order of predecessor gates. Up until now, a Boolean circuit is merely a lifeless construct. The following definition defines the *function computed by a Boolean circuit*. In this text, we speak of the circuit performing some computation when we actually, more technically correct, mean the evaluation of the associated function defined below.

Definition 3.3 ([Vol99], Definition 1.7)**.** Let $C = (V, E, \alpha, \beta, \omega)$ be a Boolean circuit over a basis B with n input and m output gates For a $v \in V$ and arbitrary Boolean values $a_1, \ldots, a_n$, we define a function $\text{val}_v \colon \{0,1\}^n \to \{0,1\}$ as follows:

1. Let v have fan-in 0. If $\beta(v) = x_i$ for some i with $1 \leq i \leq n$, then $\text{val}_v(a_1, \ldots, a_n) = a_i$. If $\beta(v) = b$ for some Boolean constant from B, then $\text{val}_v = b$.
2. Let v have fan-in k with $k > 0$ and $v_1, \ldots, v_k$ be the predecessor gates of v ordered in a way that $\alpha(v_1) < \ldots < \alpha(v_k)$. In this case $F := \beta(v)$ is a Boolean function or family of Boolean functions from B by Definition 3.2 and

$$\text{val}_v(a_1, \ldots, a_n) = F(\text{val}_{v_1}(a_1, \ldots, a_n), \ldots, \text{val}_{v_k}(a_1, \ldots, a_n)).$$

For each i with $1 \leq i \leq m$ let v_i be the unique gate for which $\omega(v_i) = y_i$. The *function computed by* C, $f_C \colon \{0,1\}^n \to \{0,1\}^m$, is defined as

$$f_C(a_1, \ldots, a_n) = (\text{val}_{v_1}(a_1, \ldots, a_n), \ldots, \text{val}_{v_m}(a_1, \ldots, a_n)).$$

Definition 3.4 ([Vol99], Definition 1.11)**.** Let $C = (V, E, \alpha, \beta, \omega)$ be a Boolean circuit over a basis B. The *size* of C is the number of gates in V that are not input gates, i.e. $\left|\{v \in V \mid \beta(v) \in B\}\right|$, and the *depth* of C is the longest path between any input and output gate, i.e. the longest directed path in the graph (V, E).

A Boolean circuit differs from other computational models in the fundamental aspect that it works on a fixed number of inputs. In order to solve problems, whose instances all have different sizes, we need to construct a *family* $\{C_n\}_{n\in\mathbb{N}}$ of circuits, where each circuit C_n in that family solves the problem for instances of one specific size n.

3.2 Symmetric Functions

Definition 3.5 ([Vol99], Definition 1.8)**.** A Boolean function $f\colon \{0,1\}^* \to \{0,1\}^*$ is *symmetric*, if $f(x_1) = f(x_2)$ for any inputs x_1, x_2 having the same number of bits that are equal to 1.

In other words, a Boolean function is symmetric, if its value depends only on the number of ones of the input. We call a gate *symmetric* if it computes a symmetric function. Thus, the output of a symmetric gate is determined by the sum of its inputs.

Example. We define the function MOD_m to evaluate to 0 if $\sum_{i=1}^{n} x_i$ divides m, where $x_1, \ldots x_n$ is the given input of length n, and 1 otherwise. This function is symmetric since its output only depends on the number of ones in the input. Another example is the *threshold* function θ_t, where $\theta_t(x_1, \ldots, x_n) = 1$ if and only if $\sum_{i=1}^{n} x_i \geq t$. The functions MOD_2 and $\theta_{n/2}$ are also referred to as *parity* and *majority*. Majority is also denoted as MAJ.

Definition 3.6 ([MT98], Definition 2.2)**.** Let SYM denote the set of functions that can be computed by a family of polynomial-size circuits of depth 1 consisting of only symmetric gates.

We present two more statements about symmetric functions that we use later.

Lemma 3.7 ([MT98], Proposition 2.4)**.** *Suppose that f is a function defined on an input $x = (x_1, \ldots x_m)$ as $f(x) = h(\sum_{i=1}^{m} w_i x_i)$, where $w_i \in \mathbb{Z}$ and $h\colon \mathbb{Z} \to \{0,1\}$. In this case f can be computed by a circuit consisting of a single symmetric gate having fan-in $m \cdot \max|w_i|$.*

Lemma 3.8 ([MT98], Corollary 2.6)**.** *Every function that can be computed by k symmetric gates of fan-in m can be computed by a single symmetric gate having fan-in $(2m)^k$.*

3.3 NC, AC and TC

Definition 3.9 ([AB07], Definition 6.24)**.** For $d \in \mathbb{N}$, the class NC^d contains all languages that can be decided by a family of Boolean circuits $\{C_n\}$ of fan-in 2, over the standard basis, where the size of C_n is polynomial in n and C_n has depth $\mathcal{O}(\log^d(n))$. We denote the union $\bigcup_{i\in\mathbb{N}_0} \mathsf{NC}^i$ as NC.

Relaxing the condition of fan-in 2 yields the similarly defined class AC.

Definition 3.10 ([AB07], Definition 6.25)**.** For $d \in \mathbb{N}$, the class AC^d contains all languages that can be decided by a family of Boolean circuits $\{C_n\}$ of *unbounded fan-in*, over the standard basis, where the size of C_n is polynomial in n and C_n has depth $\mathcal{O}(\log^d n)$. We denote the union $\bigcup_{i\in\mathbb{N}_0} \mathsf{AC}^i$ as AC.

AC^0 circuits cannot compute parity, majority, MOD_m for any constant m, and the threshold function θ_t for any $t \notin (\log n)^{\mathcal{O}(1)}$ [MT98, p. 58]. Adding majority gates, i.e. gates that evaluate to 1 if and only if the majority of their inputs are 1, to AC^0 yields the class TC^0.

Definition 3.11 ([Vol99], Definition 4.34)**.** For $d \in \mathbb{N}$, the class TC^d contains all languages that can be decided by a family of Boolean circuits $\{C_n\}$ over the standard basis with *additional majority gates*, of unbounded fan-in, where the size of C_n is polynomial in n and C_n has depth $\mathcal{O}(\log^d n)$. We denote the union $\bigcup_{i \in \mathbb{N}_0} \mathsf{TC}^i$ as TC.

As shown in [Haj+93, Proposition 2.1], every symmetric function can be computed by a TC^0 circuit. Its easy to see that every TC circuit can be realized using only majority and "not" gates:

$$\begin{aligned} \mathrm{AND}(x_1, x_2) = 1 &\iff x_1 = 1 \wedge x_2 = 1 \iff \mathrm{MAJ}(x_1, x_2, 0) = 1 \quad \text{and} \\ \mathrm{OR}(x_1, x_2) = 1 &\iff x_1 = 1 \vee x_2 = 1 \iff \mathrm{MAJ}(x_1, x_2, 1) = 1. \end{aligned}$$

Theorem 3.12 ([Vol99], Corollary 4.35)**.** $\mathsf{NC}^0 \subsetneq \mathsf{AC}^0 \subsetneq \mathsf{TC}^0 \subseteq \mathsf{NC}^1$

An important open question in circuit complexity theory is whether $\mathsf{TC}^0 \stackrel{?}{=} \mathsf{NC}^1$, thus whether the last inclusion above is strict.

3.4 Constant-Depth Circuits

The classes NC^0, AC^0 and TC^0 are all referred to as constant-depth circuit classes, since the depth of the circuits contained in these classes is bound by $\mathcal{O}(\log^0(n)) = \mathcal{O}(1)$. Since later we are interested in designing circuits of small depth, we want to refine these constant-depth circuit complexity classes even further.

Definition 3.13 ([MT98], p. 58)**.** We denote as NC^0_d, AC^0_d, and TC^0_d the sets of functions computed by, respectively, NC^0, AC^0, and TC^0 circuits having depth exactly d.

Every output gate of a NC^0 circuit computes a function of a constant number of inputs. Transforming this function into disjunctive normal form allows us to compute it using circuit of depth two and fan-in two.

Definition 3.14 ([MT98], p. 58 and Definition 2.12)**.** Denote as Σ_d and Π_d the sets of functions computed by AC^0_d circuits whose output gates consist entirely of OR and AND gates, respectively. Let Δ_d be the set of function computed by both Σ_d and Π_d circuits, i.e. $\Delta_d = \Sigma_d \cap \Pi_d$.

3.5 A Calculus for the Composition of Circuit Classes

Definition 3.15 ([MT98], Definition 2.7)**.** Let Γ_1 and Γ_2 be sets of Boolean functions. We denote as $\Gamma_1 \circ \Gamma_2$ the set of functions f of the form $f = f_1 \circ f_2$, where $f_1 \in \Gamma_1$ and $f_2 \in \Gamma_2$.

Although the above definition does not make any reference to Boolean circuits, if Γ_1 and Γ_2 are circuit complexity classes, then $\Gamma_1 \circ \Gamma_2$ is the set of circuits constructed from a circuit in Γ_2 whose output is fed into a circuit in Γ_1. More precisely, if $g \in \Gamma_1$ is computed by a family of circuits $\{G_n\}$ and $h \in \Gamma_2$ is computed by a family of circuits $\{H_n\}$, then a function f is in $\Gamma_1 \circ \Gamma_2$ if it is computed by a family of circuits $\{F_n\}$ that, for an input x

of length n, evaluate H_n for x, producing an output y of length m and then evaluate G_m for y, thus computing the function $g \circ h$ in two stages. [MT98, Section 2.4]

We present some properties exhibited by the composition of circuit classes of constant depth. The proofs of most of these statements is beyond the scope of this text; the first one, however, is straight-forward.

Lemma 3.16. $\Delta_j \circ AC_k^0 = \Delta_{j+k}$

Proof. A circuit is in Δ_j if it can be computed by both AC_j^0 circuits whose output gates consists entirely of AND gates as well as AC_j^0 circuits whose output gates consist entirely of OR gates. Adding a AC_k^0 circuit above, changes the total depth to $j + k$ but keeps the output layer untouched. The resulting circuit is in Δ_{j+k}. □

Lemma 3.17 ([MT98], Proposition 2.13)**.** $\Delta_j \circ \Delta_k = \Delta_{j+k-1}$ *for every* $j, k \geq 1$.

Lemma 3.18 ([MT98], p. 63)**.** $\mathsf{NC}^0 \circ \Delta_d = \Delta_d$ *for all* $d \geq 2$.

Lemma 3.19 ([MT98], p. 65)**.** $\mathsf{NC}^0 \subseteq \Delta_1 \circ \mathsf{NC}_1^0$

Lemma 3.20 ([MT98], Proposition 2.15)**.** $\mathsf{SYM} \subseteq \Delta_1 \circ \mathsf{NC}_1^0 \circ \mathsf{TC}_1^0$

3.6 Uniformity

In general, we do not demand circuits of one family to have any similarity in structure. As opposed to e.g. Turing machines, Boolean circuits are therefore called a *non-uniform* model of computation, whereas the former are called *uniform*.

A consequence of the non-uniformity is that we can define a family of circuits $\{H_n\}_{n\in\mathbb{N}}$ which decides a unary version of the classically undecidable halting problem by defining each H_n to output 1 if the string 1^n is the unary encoding of tuple (M, x) such that M is a TM and halts on x. Although this circuit family can be defined, there exists no procedure, let alone one that runs in polynomial time, which outputs a description of an actual circuit H_n for a given n. For this reason, we are most often interested in *uniform* circuit families. *Uniform* means that the structure of a circuit in a family is computable under some time or space constraints. [AB07, Section 6.2]

Definition 3.21 ([AB07], Definition 6.14)**.** A circuit family $\{C_n\}_{n\in\mathbb{N}}$ is *logspace-uniform* if there is a function computable in $\mathcal{O}(\log n)$ space that maps the string 1^n to a description of the circuit C_n.

According to [AB07, p. 112], a circuit family $\{C_n\}_{n\in\mathbb{N}}$ is logspace-uniform if and only if the following questions can be answered by a computation using only logarithmic space:

- What is the size of the circuit C_n?
- Is a node $u \in V$ a predecessor of a node $v \in V$?
- How many predecessors does a node have?
- Which function or input is a gate associated with, i.e. what is the value of $\beta(v)$ for some $v \in V$?

Since these computations only use logarithmic space, they can be performed in polynomial time.

For classes of "small" circuits, however, even the logspace-uniform version of these classes contain circuits whose computational power seems to stem from the process of constructing them and not the circuit itself [HAM02, Section 2.2]. For this reason, the even more restrictive notion of DLOGTIME-uniformity is often used when considering those circuits. DLOGTIME-uniformity requires that questions like the ones listed above can be answered in a logarithmic amount of time for any circuit C_n of a family. Logarithmic computation time only makes sense for machines that can access all cells in constant time *(random access machines*) since a classical TM that is given an input of size n would not even be able to read each bit of the input in $\mathcal{O}(\log n)$ time.

We barely scratch the surface of circuit uniformity. Formally proving that the circuits we look at in this chapter satisfy certain uniformity conditions is out of the scope of this work. From now on, it is therefore assumed that all presented circuits are uniform for a certain notion of uniformity and it is feasible to compute functions that answer the questions listed above. In some sense, this is implied if we explicitly describe the structure of a circuit.

3.7 Carry-Lookahead Addition

We describe the construction of a circuit that computes the sum of two n-bit integers using the *carry-lookahead* method. The construction loosely follows the detailed exposition in [HPA12][Appendix J.8] and is adapted to follow our notation.

Circuits for binary addition are commonly constructed from simple building blocks such as *half adders* and *full adders*. Both produce two outputs: a sum bit S_i and a carry bit C_i. The difference being that the full adder has a third input gate for a carry-in bit. Hence, the full adder is defined by the following equations:

$$S_i = x_i + y_i + C_i \bmod 2 \tag{3.1}$$

$$C_{i+1} = x_i y_i + x_i C_i + y_i C_i \bmod 2. \tag{3.2}$$

And we can realize the sum and carry bit using the gates introduced in this chapter:

$$S_i = \mathrm{MOD}_2(x_i, y_i, C_i) \quad \text{and} \quad C_{i+1} = \mathrm{MOD}_2(\mathrm{AND}(x_i, y_i), \mathrm{AND}(x_i, C_i), \mathrm{AND}(y_i, C_i)).$$

In order to construct a circuit for adding two n-bit numbers we face the problem of propagating the carries from one full adder to the next. The simplest solution to this problem is the so-called *carry-ripple* method, which links n full adders in a chain by connecting the carry-out of one adder to the carry-in of the next. The carry-in of the last full adder in this chain might depend on the results of all preceding additions, e.g. in the addition of the binary number $111\ldots11$ and $000\ldots01$. Hence, we observe the key weakness of the carry-ripple method, being that the last full adder has to wait until all previous carry bits have been computed. We will not explain the carry-ripple construction in further detail, instead we are interested in another method called *carry-lookahead*, that improves upon this problem. We can rewrite Eq. (3.2) as

$$C_{i+1} = G_i + P_i C_i, \tag{3.3}$$

where $G_i = x_i y_i$ and $P_i = x_i + y_i$. If $G_i = 1$, then $C_{i+1} = 1$, so a carry is *generated*. And if $P_i = 1$, then if $C_i = 1$, the carry is *propagated* and $C_{i+1} = 1$. Successively inserting Eq. (3.3) into itself yields

$$C_i = G_{i-1} + P_{i-1}G_{i-2} + P_{i-1}P_{i-2}G_{i-3} + \ldots P_{i-1}P_{i-2}\ldots P_1 G_0 + P_{i-1}P_{i-2}\ldots P_1 P_0 C_0. \tag{3.4}$$

For $i > j$, let $R_{i,j}$ denote $\mathrm{AND}(P_{i-1}, \ldots, P_{j+1}, G_j)$. $R_{i,j} = 1$ if the carry from position j is propagated to position i. By requiring the initial carry and propagation bits to be zero, i.e. $C_0 = G_0 = 0$, the last two terms of Eq. (3.4) vanish and we can rewrite the equation as

$$C_i = \mathrm{OR}(R_{i,(i-1)}, \ldots, R_{i,1}).$$

Alternatively, define H_i as $\mathrm{AND}(\overline{x_i}, \overline{y_i})$. If $H_i = 1$, then $C_{i+1} = 0$, hence the carry-in C_i is consumed. Now for $i > j$, let $Q_{i,j}$ denote $\mathrm{AND}(P_{i-1}, \ldots, P_{j+1}, H_j)$. If $Q_{i,j} = 1$, a carry bit coming into position $j + 1$ would be propagated to position i since all of the propagation bits $P_{i-1}, \ldots, P_{j+1}$ are true. But since $H_j = 1$, the carry is consumed by position j. Thus, if $Q_{i,j} = 1$, position j prevents position i from receiving a carry. The carry bit C_i is set if no position prevents position i from receiving a carry bit, i.e.

$$C_i = \mathrm{AND}(\overline{Q_{i,(i-1)}}, \ldots, \overline{Q_{i,0}}),$$

where we set H_0 to 1.

3.8 Complexity of Addition

Problem 3.22. ADD is the problem of computing the $(n + 1)$-bit sum of two n-bit numbers.

Lemma 3.23 ([MT98], Theorem 3.1)**.** $\mathrm{ADD} \in \Delta_2 \circ \mathsf{NC}^0_1$

Proof. We use the carry-lookahead method presented above. The computation of the carry bit C_i can be realized in a circuit consisting of three levels, more precisely in $\Delta_2 \circ \mathsf{NC}^0$. In the first level, the propagation bits $P_{i,j}$ are computed using at most n MOD_2 gates, thus the first level is in NC^0. In the next level we compute either $R_{i,j}$ or $Q_{i,j}$ using AND gates, hence the second level is in Σ_1. The carry bit can than be computed either as

$$C_i = \mathrm{OR}(R_{i,(i-1)}, \ldots, R_{i,1}) \quad \text{or} \quad C_i = \mathrm{AND}(\overline{Q_{i,(i-1)}}, \ldots, \overline{Q_{i,1}}).$$

We immediately see that the second variant is in Π_2, since it is computed by a circuit consisting of one level of NOT gates followed by a level of AND gates, or more precisely one AND gate. We can artificially increase the depth of the first variant by adding a second layer consisting of a single OR gate that has as inputs C_i, as defined above, and a constant 0. Hence, the first variant is in Σ_2. Level three is in both Σ_2 as well as Π_2 and therefore in Δ_2, which is equal to Δ_1 by Lemma 3.17. Level two and three combined are in $\Delta_1 \circ \Sigma_1$ and thus in Δ_2 by Lemma 3.16 because $\Sigma_1 \subseteq \mathsf{AC}^0_1$. Finally we compute each bit S_i of the sum from the input bits x_i and y_i and the carry bit as $S_i = \mathrm{MOD}_2(x_i, y_i, C_i)$, which is in NC^0. In summary the carry-lookahead circuit is in $\mathsf{NC}^0 \circ \Delta_2 \circ \mathsf{NC}^0 = \Delta_2 \circ \mathsf{NC}^0$, where the equality comes from Lemma 3.18. By Lemma 3.19 $\mathsf{NC}^0 \subseteq \Delta_1 \circ \mathsf{NC}^0_1$, thus $\Delta_2 \circ \mathsf{NC}^0 \subseteq \Delta_2 \circ \Delta_1 \circ \mathsf{NC}^0_1 = \Delta_2 \circ \mathsf{NC}^0_1$, where the equality follows from Lemma 3.17. □

3.9 Iterated Addition

Problem 3.24. ITADD is the problem of computing the $(n + \lceil \log n \rceil)$-bit sum of n n-bit numbers.

Corollary 3.25 ([MT98], Theorem 3.3)**.** $\mathrm{ITADD} \in \Delta_2 \circ \mathsf{NC}^0_1 \circ \mathsf{TC}^0_1$.

Proof. Let $x_1, \ldots, x_n$ be the input numbers given as $x_i = x_{i,n}, \ldots, x_{i,1}$ for $i \in \{1, \ldots, n\}$. Divide each input number x_i into m portions of length l, where $l = \lceil \log n \rceil$ and $m = \lceil n/l \rceil$. Let S_k be the sum of the k-th blocks of all input numbers.

$$S_k = \sum_{i=1}^{n} \sum_{j=1}^{l} x_{i,\,(k-1)\cdot l+j}\; 2^{j-1}$$

$$S = \sum_{k=1}^{m} S_k\; 2^{(k-1)l}$$

Since the maximum value of each block, when seen as a separate l bit number, is $2^l - 1$, we see that $S_k \leq n(2^l - 1)$ for each $k \in \{1, \ldots, m\}$. Each S_k has a length of at most $2l$ bits because

$$n(2^l - 1) \leq n\; 2^l = 2^{l+\log n} \leq 2^{2l}.$$

We can split each S_k into a lower- and higher-order half both having length l, which we denote as L_k and H_k. Thus, $S_k = H_k 2^l + L_k$. Denote as S_H and S_L the result of concatenating the numbers H_k and L_k, respectively. Therefore,

$$\begin{aligned} S_H + S_L &= \sum_{k=1}^{m} H_k 2^{kl} + \sum_{k=1}^{m} L_k 2^{(k-1)l} \\ &= \sum_{k=1}^{m} \Big(H_k 2^{(k-1)l+l} + L_k 2^{(k-1)l} \Big) \\ &= \sum_{k=1}^{m} (H_k 2^l + L_k) 2^{(k-1)l} \\ &= \sum_{k=1}^{m} S_k 2^{(k-1)l} = S \end{aligned}$$

Each bit of S_H and S_L is a function of one of the S_k, and S_k itself is a weighted sum of the input numbers $x_1, \ldots, x_n$. We can thus apply Lemma 3.7, yielding that every bit of S_H and S_L can be computed by a circuit consisting of a single symmetric gate. Since the addition of the two numbers S_H and S_L is in $\Delta_2 \circ \mathsf{NC}^0_1$ by Lemma 3.23, ITADD is in $\Delta_2 \circ \mathsf{NC}^0_1 \circ \mathsf{SYM}$. A circuit in $\mathsf{NC}^0_1 \circ \mathsf{SYM}$ computes a function of symmetric gates and can therefore be computed by a single symmetric gate according to Lemma 3.8. Thus, ITADD $\in \Delta_2 \circ \mathsf{SYM}$. Lemma 3.20 states that $\mathsf{SYM} \subseteq \Delta_1 \circ \mathsf{NC}^0_1 \circ \mathsf{TC}^0_1$, therefore

$$\text{ITADD} \in \Delta_2 \circ \mathsf{SYM} \subseteq \Delta_2 \circ \Delta_1 \circ \mathsf{NC}^0_1 \circ \mathsf{TC}^0_1.$$

By Lemma 3.17, $\Delta_2 \circ \Delta_1 = \Delta_{2+1-1} = \Delta_2$. Hence follows the claim. □

Definition 3.26. BitITADD is the language of tuples $(X_1, \ldots, X_n, j, 1^n)$ such that X_i is a n-bit number for each $i \in \{1, \ldots, n\}$ and the j-th bit of the sum $\sum_{i=1}^{n} X_i$ is 1.

Lemma 3.27. BitITADD $\in \mathsf{PH}^{\mathsf{PP}}$

Proof. We show in Corollary 3.25 that the problem ITADD of adding n n-bit numbers can be solved by a circuit in $\Delta_2 \circ \mathsf{NC}^0_1 \circ \mathsf{TC}^0_1$. Such a circuit consists of four layers: a layer of majority gates, one layer of AND and OR gates of bounded fan-in, followed by two layers of AND and OR gates of unbounded fan-in. We show how the task of determining the value of a single output gate of such circuits can be realized in $\mathsf{PH}^{\mathsf{PP}}$.

Let the predicate $\mathrm{Con}(u, v)$ be true if and only if gate u is connected to gate v. Furthermore, let Lay_i denote the set of gates in layer i. Note that layer 0 consists entirely of input nodes.

Under the assumption that these circuits are uniform, for a suited definition, we are able to answer questions about their structure in polynomial time. E.g. we are can evaluate the Con or list all gates of layer i.

Let Val_i be the set of gates in layer i that evaluate to 1 and let the unary predicate $\mathrm{Val}_i(v)$ be true if and only if $v \in \mathrm{Val}_i$. If v is an OR gate, the output of v is 1 if there exists a predecessor gate in the preceding layer that evaluates to 1, i.e.

$$\mathrm{Val}_i(v) \iff \exists u \in \mathrm{Lay}_{i-1} \colon \mathrm{Con}(u, v) \wedge \mathrm{Val}_{i-1}(u).$$

If v is an AND gate, the output of v is 1 if and only if all of its predecessor gates in the preceding layer evaluate to 1, i.e.

$$\mathrm{Val}_i(v) \iff \forall u \in \mathrm{Lay}_{i-1} \colon \mathrm{Con}(u, v) \to \mathrm{Val}_{i-1}(u).$$

By the construction of the ITADD circuit, the number of gates in each layer is at most exponential in n. Therefore, each gate can be represented by a polynomial-length string and we can quantify over the set $\{0,1\}^{p(n)}$ for some polynomial p instead of quantifying over a set of gates, in the above formula. Additionally, all predicates except $\mathrm{Val}_{i-1}(u)$ can be evaluated in polynomial time. It follows, that $\mathrm{Val}_i \in \mathsf{PH}^{\mathrm{Val}_{i-1}}$ for all $i \in \{2, 3, 4\}$, by the definition of the polynomial hierarchy.

The first layer consist entirely of majority gates. The output of a majority gate is 1 if and only if the majority of its inputs are 1. Let f be the function that counts the number of predecessor gates of a majority gate v that evaluate to 1. Formally:

$$f(v) = \left|\left\{u \in \mathrm{Lay}_0 \mid \mathrm{Con}(u, v) \wedge \mathrm{Val}_0(u)\right\}\right|.$$

Instead of taking u from a set of gates, we can identify u with a string from $\{0,1\}^{p(n)}$ for some polynomial p. Furthermore, the predicate Con is computable in polynomial time because of the assumed uniformity and Val_0 is given by the input. Thus, the function f is in $\#\mathsf{P}$. The total number of predecessor gates of v can also be determined in polynomial time because of the assumed uniformity. If the value $f(v)$ is known, a polynomial-time TM can decide if the majority of the inputs of v are 1. Hence, $\mathrm{Val}_1 \in \mathsf{P}^{\#\mathsf{P}}$ and therefore $\mathrm{Val}_1 \in \mathsf{P}^{\mathsf{PP}}$, since $\mathsf{P}^{\#\mathsf{P}} = \mathsf{P}^{\mathsf{PP}}$ by Lemma 2.12.

In summary, $\mathrm{Val}_4 \in \mathsf{PH}^{\mathrm{Val}_3}$, $\mathrm{Val}_3 \in \mathsf{PH}^{\mathrm{Val}_2}$, $\mathrm{Val}_2 \in \mathsf{PH}^{\mathrm{Val}_1}$ and $\mathrm{Val}_1 \in \mathsf{P}^{\mathsf{PP}}$. Combining these findings, we see that

$$\mathrm{Val}_4 \in \mathsf{PH}^{\mathsf{PP}}$$

since $\mathsf{PH}^{\mathsf{PH}} = \mathsf{PH}$ (Lemma 2.8) and $\mathsf{PH}^{\mathsf{P}} = \mathsf{PH}$ (Corollary 2.9). BitITADD is equal to deciding whether the j-th output gate is in Val_4. □

Problem 3.28. LOGITADD is the problem of computing the sum of $\log n$ many n-bit numbers.

Lemma 3.29 ([Vol99], Theorem 1.21)**.** LOGITADD *can be solved by a uniform* AC^0 *circuit.*

Lemma 3.30. *The problem of deciding whether the j-th output gate of a given uniform* AC^0 *circuit evaluates to 1 is in* PH*.*

Proof. AC^0 circuits have constant depth but the fan-in of each gate is unbounded. Without loss of generality, we can assume that the given circuit is made up of a constant number of layers. The proof is analogous to the first part of the proof of Lemma 3.27: A gate v is in Val_i if it is in layer i and evaluates to 1. Using predicates like $\mathrm{Con}(u, v)$, which can be decided in polynomial time sine we only consider uniform circuits, we show that $\mathrm{Val}_i \in \mathsf{PH}^{\mathrm{Val}_{i-1}}$ for each layer i. Since $\mathsf{PH}^{\mathsf{PH}} = \mathsf{PH}$, the values of gates in the output layer can be decided in PH. □

4. Number Theoretical Notions

4.1 Chinese Remainder Representation

The *Chinese remainder theorem* is a basic result from number theory. A statement of this theorem was mentioned by the Chinese mathematician Sun Tzu who lived between 544 an 496 B.C.

Theorem 4.1 (Chinese remainder theorem – [HW79], Theorem 121)**.** *The system of congruences*

$$\begin{aligned} X &\equiv a_1 \pmod{m_1} \\ &\vdots \\ X &\equiv a_k \pmod{m_k} \end{aligned}$$

has a unique solution if the moduli $m_1, \ldots, m_k$ are pairwise coprime.

Definition 4.2. For $n \in \mathbb{N}$, let $\mathcal{P}_n$ be the set of all odd primes less than n and $P_n = \prod_{p \in \mathcal{P}_n} p$.

The Chinese remainder theorem implies that a nonnegative integer can be represented as a list of remainders modulo some primes, which is sometimes referred to as *Chinese remainder representation*. This representation has some interesting properties that we apply in Chapter 6.

Lemma 4.3. *For $n \in \mathbb{N}$, every nonnegative integer X smaller than P_n can be represented uniquely as a list $(x_p)_{p \in \mathcal{P}_n}$, where $x_p \equiv X \pmod{p}$.*

Proof. Let $p_1, \ldots, p_k$ denote the elements of $\mathcal{P}_n$. By the Chinese remainder theorem (4.1), the system of congruences

$$\begin{aligned} X &\equiv a_1 \pmod{p_1} \\ &\vdots \\ X &\equiv a_n \pmod{p_k} \end{aligned}$$

has a unique solution modulo P_n because $p_1, \ldots, p_k$ are prime and therefore pairwise coprime. In other words $X \mapsto (X \bmod p_1, \ldots, X \bmod p_k)$ defines a ring isomorphism and

$$\mathbb{Z}/P_n\mathbb{Z} \cong \mathbb{Z}/p_1\mathbb{Z} \times \ldots \times \mathbb{Z}/p_k\mathbb{Z}.$$

Thus, we can represent any integer less than N as a list $(x_p)_{p\in\mathcal{P}_n}$, where $x_p = X \bmod p$. □

Lemma 4.4. *Let X be a nonnegative integer smaller than P_n and let $h_{p,n}$ denote the modular inverse of P_n/p mod p, then*

$$X \equiv \sum_{p\in\mathcal{P}_n} x_p h_{p,n} P_n/p \pmod{P_n}.$$

Proof. By the Chinese remainder theorem, it suffices to show that the congruence

$$\sum_{p\in\mathcal{P}_n} x_p h_{p,n} P_n/p \equiv x_q \pmod{q}$$

is satisfied for every odd prime number q less than P_n. Since $h_{p,n}$ is the modular inverse of P_n/p modulo p and every odd prime number q' with $q' < P_n$ and $q' \neq p$ divides P_n/p, the following two congruences hold:

$$h_{p,n}P_n/p \equiv 1 \pmod{p} \quad \text{and} \quad h_{p,n}P_n/p \equiv 0 \pmod{q'}.$$

Therefore

$$\sum_{p\in\mathcal{P}_n} x_p h_{p,n} P_n/p \equiv \sum_{p\in\mathcal{P}_n} (x_p \bmod q)(h_{p,n}P_n/p \bmod q) \equiv x_q \pmod{q}. \qquad \square$$

Definition 4.5. For $x \in \mathbb{R}$, let $\lfloor x \rfloor$ and $\{x\}$ be the *integral* and *fractional* part of x, respectively.

For example, $\{\pi\} = 0.14159\ldots$ and $\lfloor \pi \rfloor = 3$. We observe the following basic properties concerning the integral and fractional part.

Observation 4.6. *For every $x \in \mathbb{R}$:*

1. $\lfloor x \rfloor \in \mathbb{N}$
2. $0 \leq \{x\} < 1$
3. $x = \lfloor x \rfloor + \{x\}$
4. $\lfloor x \rfloor = 0 \iff x < 1$
5. $x < 1 \implies \{x\} = x$.

Corollary 4.7. *The fractional part of $\sum_{p\in\mathcal{P}_n} x_p h_{p,n}/p$ is given by X/P_n for every $X \in \mathbb{N}_0$ with $X < P_n$.*

Proof. Let us denote the number $\sum_{p\in\mathcal{P}_n} x_p h_{p,n}/p$ as $\widetilde{X}$. By Lemma 4.4, X is congruent to $\sum_{p\in\mathcal{P}_n} x_p h_{p,n} P_n/p$ modulo P_n, i.e. there exists an integer q such that

$$X = q \cdot P_n + \sum_{p\in\mathcal{P}_n} x_p h_{p,n} P_n/p.$$

Dividing both sides by P_n and rearranging yields

$$q = \sum_{p \in \mathcal{P}_n} (x_p h_{p,n}/p) - X/P_n.$$

After decomposing each summand into its integral and fractional part we are left with

$$q = \lfloor \widetilde{X} \rfloor + \{\widetilde{X}\} - \lfloor X/P_n \rfloor - \{X/P_n\}.$$

Since $X < P_n$, $X/P_n < 1$ and $\lfloor X/P_n \rfloor = 0$. Consequently $q = \lfloor \widetilde{X} \rfloor + \{\widetilde{X}\} - \{X/P_n\}$. Since q and $\lfloor \widetilde{X} \rfloor$ are integers, the fractional parts on the right side must cancel out, hence $\{\widetilde{X}\} = \{X/P_n\}$. Since $X/P_n < 1$, $\{\widetilde{X}/P_n\} = X/P_n$. □

Definition 4.8. For $n \in \mathbb{N}$, let $N(n)$ denote the value 2^{n^2}. In cases where no confusion can arise, we simply write N instead of $N(n)$.

According to the above definition, $\mathcal{P}_N$ is the product of all odd primes less than 2^{n^2}.

Lemma 4.9. $2^{2^n} < P_N < 2^{2^{n^2+1}}$ *for large enough* $n \in \mathbb{N}$.

Proof. For any integer n, the prime-counting function π gives the number of primes not exceeding n. The prime number theorem states that $\pi(n) \sim \frac{n}{\log n}$, i.e. $\lim_{n\to\infty} \frac{\pi(n) \log n}{n} = 1$ [HW79, Theorem 6]. This implies that for all $\varepsilon > 0$ there exist an integer n_0, such that for every $n \geq n_0$

$$(1-\varepsilon)\frac{n}{\log n} \leq \pi(n) \leq (1+\varepsilon)\frac{n}{\log n}.$$

By choosing $\varepsilon = 1/2$, we see that for large enough n

$$\pi(n) \geq (1 - \frac{1}{2})\frac{n}{\log n} = \frac{n}{2 \log n}. \tag{4.1}$$

Similarly by choosing $\varepsilon = 1$, we see that for large enough n

$$\pi(n) \geq (1+1)\frac{n}{\log n} = \frac{2n}{\log n}. \tag{4.2}$$

Since every $p \in \mathcal{P}_N$ is larger than 2 and there are $\pi(N)$ primes smaller than N, we see that

$$P_N = \prod_{p \in \mathcal{P}_N} p > 2^{\pi(N)}.$$

For large enough n, inequality 4.1 implies that

$$\pi(N) \geq \frac{N}{2 \log N} = \frac{2^{n^2}}{2n^2} \geq \frac{2^{n^2}}{2^{n^2-n}} = 2^{n^2-n^2+n} = 2^n$$

were the second inequality follows from $2n^2 \leq 2^{n^2-n}$ for $n \geq 5$ which can be shown using induction. Therefore $P_N > 2^{2^n}$. Similarly it follows that

$$P_N < N^{\pi(N)} = \left(2^{n^2}\right)^{\pi(N)} \leq \left(2^{n^2}\right)^{\left(\frac{2N}{\log N}\right)} = 2^{\left(n^2 \frac{1}{n^2} 2^{n^2+1}\right)} = 2^{2^{n^2+1}}$$

for large enough n, by inequality 4.2. □

4.2 Lagrange's Theorem

Definition 4.10. Let G be a finite group. For any $g \in G$, the set $\langle g \rangle = \{g^n \mid n \in \mathbb{Z}\}$ is called the *cyclic subgroup* generated by g. The *order* of an element $g \in G$, denoted as $\mathrm{ord}(g)$, is the number of elements in the cyclic subgroup generated by g or, equivalently, the smallest positive integer i for which $g^i = 1$.

Definition 4.11. Let G be a finite group of order m, i.e. $|G| = m$. If there exists a $g \in G$ that has order m, then $\langle g \rangle = G$. In this case we call g a *generator* of G and say that G is *cyclic*.

Theorem 4.12 (Lagrange's Theorem)**.** *If G is a finite group and H is a subgroup of G, then $|H|$ divides $|G|$.*

Proof. Let $gH = \{gh \mid h \in H\}$ be the so-called *left coset* of H for some $g \in G$. We show that every such left coset has the same number of elements as H and that the left cosets of H form a partition of G. As G can be divided into equally sized portions of $|H|$ elements, the claim follows.

For every $g \in G$, consider the functions

$$f_g \colon H \to gH,\ h \mapsto gh \quad \text{and} \quad f'_g \colon gH \to H,\ h \mapsto g^{-1}h.$$

It is easy to see that f'_g is the inverse of f_g, hence f_g is a bijection. Therefore every left coset of H has the same number of elements as H.

Suppose g_1H and g_2H are left cosets and $g_1H \cap g_2H \neq \emptyset$. Then there are $h_1, h_2 \in H$ such that $g_1h_1 = g_2h_2$. Rearranging yields $g_1 = g_2h_2h_1^{-1}$. Thus, for every $h \in H$,

$$g_1h = g_2h_2h_1^{-1}h \in g_2H.$$

Since h is arbitrary, $g_1H \subseteq g_2H$. The inclusion $g_2H \subseteq g_1H$ follows by the same reasoning, thus $g_1H = g_2H$. Therefore, every pair of left cosets g_1H and g_2H is either disjoint or identical. Furthermore, for every $g \in G$, $g = g \cdot 1 \in gH$ since H, as a subgroup of G, contains the neutral element 1. Hence the left cosets form a partition of G.

In conclusion the number of elements of all left cosets of H are equal to $|H|$ and add up to $|G|$. Thus, $|H|$ must divide $|G|$. □

Corollary 4.13. *Let G be a finite group, then $g^{|G|} = 1$ for every $g \in G$.*

Proof. Consider the subgroup $\langle g \rangle$ generated by some $g \in G$ and let $q = \mathrm{ord}(g) = |\langle g \rangle|$. We have $g^q = 1$. By Theorem 4.12, q divides $|G|$, i.e. there exists some $k \in \mathbb{Z}$ such that $kq = |G|$. Thus, $g^{|G|} = g^{kq} = (g^q)^k = 1^k = 1$. □

4.3 Multiplicative Inverses

When speaking of $\mathbb{Z}/n\mathbb{Z}$ for some $n \in \mathbb{N}$, we always mean the group $(\mathbb{Z}/n\mathbb{Z}, \times)$, unless stated otherwise. That is, we assume multiplication as the group operation and use multiplicative notation, i.e. xy and x^{-1} to denote the product of x and y and the inverse of x, respectively.

Definition 4.14 ([HW79], p. 52)**.** For an integer n the *Euler totient function* φ gives the number of integers not greater than and coprime to n, that is

$$\varphi(n) = \left|\{m \in \mathbb{N} \mid m < n,\ \gcd(m, n) = 1\}\right|.$$

Definition 4.15. $(\mathbb{Z}/n\mathbb{Z})^\times$ is the set of units, i.e. elements which have an inverse, of $\mathbb{Z}/n\mathbb{Z}$.

An element $a \in \mathbb{Z}/n\mathbb{Z}$ is a unit of $\mathbb{Z}/n\mathbb{Z}$ if a is coprime to n, i.e. $\gcd(a, n) = 1$. Therefore,

$$(\mathbb{Z}/n\mathbb{Z})^\times = \{x \in \mathbb{Z}/n\mathbb{Z} \mid \gcd(x, n) = 1\}.$$

The set $(\mathbb{Z}/n\mathbb{Z})^\times$ contains the neutral element. Additionally, for every unit x of $\mathbb{Z}/n\mathbb{Z}$, x^{-1} is clearly also a unit of $\mathbb{Z}/n\mathbb{Z}$. Therefore, $(\mathbb{Z}/n\mathbb{Z})^\times$ forms a group under multiplication and is referred to as the *multiplicative group of the integers modulo n.*

The greatest common divisor of zero and any nonnegative integer n is n. On the other hand, every element of $\mathbb{Z}/p\mathbb{Z}$ except zero is coprime to p for any prime p. If p is prime, the multiplicative group of the integers modulo p therefore contains all elements of $\mathbb{Z}/p\mathbb{Z}$ except zero and we obtain $\left|(\mathbb{Z}/p\mathbb{Z})^\times\right| = p - 1$.

Theorem 4.16 (Fermat-Euler-Theorem)**.** *Let a and n be coprime integers, then*

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

Proof. Let a and n be coprime integers. By definition, the order of the multiplicative subgroup of integers modulo n, $(\mathbb{Z}/n\mathbb{Z})^\times$, is $\varphi(n)$. Since a and n are coprime, $a \in (\mathbb{Z}/n\mathbb{Z})^\times$. The claim follows directly from Corollary 4.13 since raising an element to the power of the group order yields the neutral element, i.e. $a^{\varphi(n)} \equiv 1 \pmod{n}$. □

Corollary 4.17. *Let p be prime and $a \in \mathbb{Z}/p\mathbb{Z}$, then $a^{-1} \equiv a^{p-2} \pmod{p}$.*

Proof. As a prime, p is coprime to every integer smaller then itself. Furthermore, p is not coprime to itself since 1 is the only number coprime to itself. Therefore $\varphi(p) = p - 1$ and according to the Fermat-Euler theorem (4.16)

$$a^{p-1} \equiv 1 \pmod{p}$$

for any $a \in \mathbb{Z}/p\mathbb{Z}$. By rewriting the above statement as $a \cdot a^{p-2} \equiv 1 \pmod{p}$, we see that a^{p-2} is the modular inverse of a modulo p. □

4.4 The Multiplicative Group of the Integers modulo a Prime

We use the notation $n|m$ to say that nonzero integer n *divides* an integer m, i.e. there exists an integer k, such that $m = kn$.

Lemma 4.18 ([HW79], Theorem 63)**.** $\sum_{d|n} \varphi(d) = n$ *for all* $n \in \mathbb{N}$.

Proof. We follow the alternative proof for [HW79, Theorem 63] given in [HW79, Section 16.2]. Consider the set of fractions $Q = \left\{\frac{k}{n} \mid 1 \le k \le n\right\}$. Each fraction k/n can be expressed in irreducible form by cancelling out all common factors of the numerator and denominator. That is $k/n = a/d$ such that $d \mid n$ and $\gcd(a, d) = 1$ with $1 \le a \le d$. Conversely, every fraction a/d, with $d \mid n$ and $\gcd(a, d) = 1$ for $1 \le a \le d$, appears in Q. Since the number of integers a that satisfy $1 \le a \le d$ and $\gcd(a, d) = 1$ is exactly $\varphi(d)$, it follows that

$$\begin{aligned} n = |Q| &= \sum_{d|n} \sum_{\substack{1 \le a \le d \\ \gcd(a,d)=1}} 1 \\ &= \sum_{d|n} \varphi(d). \end{aligned}$$

□

Theorem 4.19 ([IR90], Theorem 4.1)**.** *For every prime p, $(\mathbb{Z}/p\mathbb{Z})^\times$ is cyclic.*

Proof. Let p be some prime. We prove the cyclicity of $(\mathbb{Z}/p\mathbb{Z})^\times$ by showing the existence of an element of order $p-1$. For any $d \in \mathbb{N}$, let $\psi(d)$ denote the number of elements of $(\mathbb{Z}/p\mathbb{Z})^\times$ having order d. Hence $\psi(p-1) \geq 1$ implies that $(\mathbb{Z}/p\mathbb{Z})^\times$ is cyclic.

Every $x \in (\mathbb{Z}/p\mathbb{Z})^\times$ generates a subgroup $\langle x \rangle$ and the number of elements in $\langle x \rangle$ is the order of x. By Theorem 4.12, $\mathrm{ord}(x)$ divides $p-1$. This implies that there exists an element of order d, i.e. $\psi(d) > 0$, if and only d divides $p-1$. The total number of elements in $(\mathbb{Z}/p\mathbb{Z})^\times$ is equal the sum of the number of elements of each order, therefore

$$p - 1 = \sum_{d=1}^{p-1} \psi(d) = \sum_{d|p-1} \psi(d).$$

Applying Lemma 4.18 for $n = p-1$ yields:

$$\sum_{d|p-1} \psi(d) = p - 1 = \sum_{d|p-1} \varphi(d).$$

And therefore

$$\sum_{d|p-1} \psi(d) - \varphi(d) = 0. \tag{4.3}$$

Next, we show that $\psi(d) \leq \varphi(d)$ for all d dividing $p-1$: Suppose there exists an integer d dividing $p-1$ such that $\psi(d) > \varphi(d)$ and let h be an element of order d. Let H be the subgroup of $(\mathbb{Z}/p\mathbb{Z})^\times$ generated by h, i.e. $H = \langle h \rangle = \{h, h^2, \ldots, h^d\}$. Thus, every element of H can be written as h^j for some $j \in \{1, \ldots, d\}$.

The order of each $h^j \in H$ is the smallest number k such that $h^{jk} \equiv 1 \pmod{d}$. The order of h^j is independent of whether we regard h^j as an element of $(\mathbb{Z}/p\mathbb{Z})^\times$ or H. Therefore, the congruence $h^{jk} \equiv 1 \pmod{p}$ also holds. By Theorem 4.12, k divides d. Thus, jk is the smallest common multiple of j and d. For any two number a and b, $ab = \mathrm{lcm}(a,b) \cdot \gcd(a,b)$. Therefore:

$$\mathrm{lcm}(j,d) \cdot \gcd(j,d) = jd \iff jk \cdot \gcd(j,d) = jd \iff k = \frac{d}{\gcd(j,d)}.$$

This implies that $k = d$ if and only if $\gcd(j,d) = 1$. Hence, the number of elements of H which have order d is equivalent to the number of elements $h^j \in H$ for which $\gcd(j,d) = 1$ which is exactly $\varphi(d)$.

Since $\psi(d) > \varphi(d)$, there must exist an element $g \in (\mathbb{Z}/p\mathbb{Z})^\times$ which has order d but is not in H. By definition, $g^d \equiv 1 \pmod{p}$. Consequently, the congruence $x^d \equiv 1 \pmod{p}$ has more than d solutions: each of the d elements of H and at least one element g which is not in H. That is, the polynomial $x^d - 1 \in \mathbb{Z}/p\mathbb{Z}[x]$ has more than d roots. This is a contradiction to the fact that every polynomial of degree d over a $\mathbb{Z}/p\mathbb{Z}$, which is a field since p is prime, has at most d roots. Therefore, $\varphi(d) - \psi(d) \geq 0$ for all d dividing $p-1$.

It follows that $\varphi(d) = \psi(d)$ for all d dividing $p-1$, because each summand in Eq. (4.3) is nonnegative but the sum is equal to zero. For every prime p, $\varphi(p-1) \geq 1$. Thus, if p is prime, $\psi(p-1) = \varphi(p-1) \geq 1$, i.e. $(\mathbb{Z}/p\mathbb{Z})^\times$ is cyclic. □

4.5 Binary Expansion of Prime Reciprocals

Prime reciprocals are rational numbers of the form $1/p$, where p is prime. The representation of any rational number q in some base r is either finite or repeating, and the length of the repeating sequence of digits in this representation is called the *period* of q. William Shanks, who lived between 1812 and 1882, spend years of his life calculating the decimal expansion of reciprocals of every prime number up to 110 000 in order to determine their period. Luckily, the following statement allows us to easily compute the bit at position j of the binary expansion of prime reciprocals. We show in Chapter 6 that this method is efficient even if j is "very large".

Lemma 4.20 ([Kak87])**.** *The j-th bit of the binary expansion of $1/p$ is 1 if and only if $2^j \bmod p$ is odd for every odd prime p.*

Proof. For any $p \in \mathbb{Z}$, the so-called d-sequence $a_1, a_2, \ldots$ of $1/p$ are the digits that represent the expansion of $1/p$ in base $r \geq 2$. The statement given in [Kak87] is the following:

$$a_i = l(r^i \bmod p) \bmod r,$$

where $l \in \mathbb{Z}$ is a number such that $l < r$ and $-l \equiv p \pmod{r}$. In this case $r = 2$ and p is an odd prime. Thus, $p \bmod r = 1$. For $l = 1$, the congruence $-l \equiv p \pmod{r}$ is satisfied, since $-1 \equiv 1 \pmod{2}$. Therefore $a_i = (2^i \bmod p) \bmod 2$. □

5. Computation over the Reals

The theory of computation over an arbitrary field or ring was introduced in 1989 by Lenore Blum, Mike Shub and Steve Smale, arising from the problem that the classical (Turing) theory of computation is inherently dependent on discrete objects, i.e. 0s and 1s, and therefore inadequate most real number algorithms from the world of numerical analysis. [Blu04]

5.1 Finite-Dimensional Machines

Definition 5.1 ([Blu+98], Definition 1)**.** A *polynomial* (or *rational*) map $g\colon R^m \to R^m$ is given by m polynomials (or rational functions) $g_i\colon R^m \to R$ for $i = 1, \ldots, m$. If g is a rational map, we assume that each g_i is given by a pair of polynomials (p_i, q_i) and $g_i(x) = p_i(x)/q_i(x)$.

Definition 5.2 ([Blu+98], Definition 1)**.** A *finite-dimensional machine* M over a ring R consist of a finite directed graph with each node being either an *input*, *computation*, *branch* or *output* node. There exists only one input node. The input node has no incoming edges and only one outgoing edge. Every other node has possibly multiple incoming edges. Computation nodes have only one outgoing edge. Branch nodes have two outgoing edges, labeled *Yes* and *No*. Output nodes have no outgoing edges. In addition, M has associated with it:

- an *input space* $\mathcal{I} = R^n$ for some $n \in \mathbb{N}$,
- a *state space* $\mathcal{S} = R^m$ for some $m \in \mathbb{N}$,
- and an *output space* $\mathcal{O} = R^l$ for some $l \in \mathbb{N}$.

Each node is equipped with maps on these spaces and a *next node assignment* as follows:

- Associated with the input node is a linear map $I\colon \mathcal{I} \to \mathcal{S}$ and a unique next node β_1.
- Each computation node η has an associated *computation map* $g_\eta\colon \mathcal{S} \to \mathcal{S}$, and a unique next node β_η. If R is a field, g_η is a rational map, otherwise g_η is a polynomial map.
- Each branch node η has an associated *branching function* which is a nonzero polynomial function $h_\eta\colon \mathcal{S} \to R$. The next node along the outgoing edge *Yes*, β_η^+, is associated with the condition $h_\eta(z) \geq 0$ and the next node along the outgoing edge

No, β_η^- is associated with $h_\eta(z) < 0$. If R is a ring or field without order, e.g. $\mathbb{C}$ or $\mathbb{Z}/n\mathbb{Z}$, we alter the above by associating β_η^+ with the condition $h_\eta(z) = 0$ and β_η^- with $h_\eta(z) \neq 0$.

- Every output node η has an associated linear map $O_\eta \colon \mathcal{S} \to \mathcal{O}$ and no next node.

It is sometimes convenient to define the computation map and next node assignment for all nodes. This can be achieved by defining the computation map to be the identity on all nodes that are not computation nodes and the next node of any output node to be itself.

The set of all coefficients of linear of rational maps that are used by a finite-dimensional machine over R is called the set of *machine constants*.

Definition 5.3 ([Blu+98], p. 44)**.** Let R be a ring and M a machine over R. Let $\mathcal{N}$ be the set of nodes of M and $\mathcal{S}$ its state space. We call the set of all node/state pairs $\mathcal{N} \times \mathcal{S}$ the *configuration space* of M.

Definition 5.4 ([Blu+98], p. 44)**.** The *computing endomorphism* $H \colon \mathcal{N} \times \mathcal{S} \to \mathcal{N} \times \mathcal{S}$ maps each node/state pair (η, x) to the unique next node/state pair (η', x'). Let $\mathcal{B} \subset \mathcal{N}$ denote the subset of branch nodes of M and $\mathcal{C}$ be $\mathcal{N} \setminus \mathcal{B}$. Then we define H as

$$H(\eta, x) = (\beta_\eta, g_\eta(x))$$

for each $\eta \in \mathcal{C}$. If $\eta \in \mathcal{B}$ is a branching node, we define H as

$$H(\eta, x) = \begin{cases} (\beta_\eta^+, g_\eta(x)) & \text{if } h_\eta(x) \geq 0 \\ (\beta_\eta^-, g_\eta(x)) & \text{if } h_\eta(x) < 0 \end{cases}$$

in the case that R has order and

$$H(\eta, x) = \begin{cases} (\beta_\eta^+, g_\eta(x)) & \text{if } h_\eta(x) \neq 0 \\ (\beta_\eta^-, g_\eta(x)) & \text{if } h_\eta(x) = 0 \end{cases}$$

in the case that R is a ring without order.

Definition 5.5 ([Blu+98], p.45)**.** Starting from an initial point $z_0 = (q_1, x^0) \in \mathcal{N} \times \mathcal{S}$, the iterated application of H generates the sequence $z_0, z_1, z_2, \ldots$, where

$$z_k = H(z_{k-1}) = H^k(z_0)$$

is called a *computation*.

Definition 5.6 ([Blu+98], p.45)**.** Let $\pi_\mathcal{N} \colon \mathcal{N} \times \mathcal{S} \to \mathcal{N}$ be the projection from the configuration space onto the set of nodes. We call the sequence of nodes $\eta_0, \eta_1, \eta_2, \ldots$, where $\eta_k = \pi_\mathcal{N}(z_k)$, the *computation path* γ_x *traversed* by M on input x.

Definition 5.7 ([Blu+98], p. 45)**.** We say that a computation *halts* if there exists an output node N, a *time* $T \in \mathbb{N}$ and some $u \in \mathcal{S}$, such that $z^T = (N, u)$. In this case the finite sequence $(z_0, z_1, \ldots, z_T) \in (\mathcal{N} \times \mathcal{S})^{(T+1)}$ is called a *halting computation*.

Definition 5.8 ([Blu+98], p.45)**.** The *halting set* of M, denoted as Ω, is the set of all inputs on which M halts. The *input-output map* $\Phi \colon \Omega \to \mathcal{O}$ maps each input in the halting set to the state on which M halts and applies the mapping to the output space, i.e.

$$\Phi(x) = O(x^T),$$

where T is the least number such that $z^T = (N, x^T)$ for some output node N or.

Definition 5.9 ([Blu+98], p.46)**.** A map $\varphi\colon X \to R^l$, where $X \subset R^n$, is computable if it is the input-output map of a finite-dimensional machine. That is, φ is computable if there exists a finite-dimensional machine M having the input space R^n, output space R^l, halting set X and input-output map φ.

Definition 5.10 ([Blu+98], p.47)**.** A set $S \subseteq R^n$ is decidable over R if the function

$$\chi\colon R^n \to R,\ x \mapsto \begin{cases} 1 & \text{if } x \in S \\ 0 & \text{otherwise} \end{cases},$$

called the *characteristic function* is decidable. Otherwise we call S undecidable over R.

When $R = \mathbb{Z}_2$, all of the above definitions are equivalent to the classical definition of a Turing machine.

5.2 The Blum-Shub-Smale Model

The Blum-Shub-Smale (BSS) Model can be seen as a generalization of the standard Turing machine where each cell of the tape can hold an element of the underlying ring or field R. Furthermore, this machine can perform specific arithmetic operations on R in constant time.

Definition 5.11 ([Blu+98], p.70)**.** Let R be a ring. We denote the disjoint union $\bigsqcup_{n\geq 0} R^n$ by R^∞.

Definition 5.12 ([Blu+98], p. 70)**.** For a ring R, we denote the *bi-infinite direct sum over* R by R_∞. Elements of R_∞ have the form

$$x = (\ldots, x_{-2}, x_{-1}, x_0 \,\#\, x_1, x_2, \ldots),$$

where $x_i \in R$ for all $i \in \mathbb{Z}$, $x_k = 0$ for $|k|$ sufficiently large, and $\#$ is a distinguished marker between x_0 and x_1. We say that x_i is the i-th *coordinate* of x for every $i \in \mathbb{Z}$.

The bi-infinite direct sum corresponds to the "working tape" of a classical Turing machine. Similarly, the distinguished marker represents the position of the "read-write head". We move the "read-write head" by defining two endomorphisms on the "tape" R_∞, *shift left* and *shift right*. Shift left moves the distinguished marker one coordinate to the right and thus each element one coordinate to the left. Shift right is the inverse operation.

Definition 5.13 ([Blu+98], p.70)**.** The two endomorphisms $\sigma_L, \sigma_R\colon R_\infty \to R_\infty$ are called *shift left* and *shift right* and are defined as $\sigma_L(x)_i = x_{i+1}$ and $\sigma_R(x)_i = x_{i-1}$ for each $i \in \mathbb{Z}$.

Before we present the definition of a *BSS machine*, we need a few more technical definitions. As in the case of finite-dimensional machines, we can define polynomial (or rational) maps from the state space onto the underlying ring. These maps are used in the branch nodes of the new machine.

Definition 5.14 ([Blu+98], p. 70)**.** Let $h\colon R^m \to R$ be a polynomial (or rational) function over R. Then h defines a *polynomial* (or *rational*) map

$$\widehat{h}\colon R_\infty \to R,\ x \mapsto h(x_1, \ldots, x_m)$$

on R_∞ of *dimension* m.

Likewise, we need to adapt the concept of polynomial (or rational functions) endomorphisms on the state space, that are used by the computation nodes.

Definition 5.15 ([Blu+98], p.70)**.** Let $g_i \colon R^m \to R$ for $i = 1, \ldots, m$ be polynomial (or rational) functions having maximum degree d over R. Then the g_i for $i = 1, \ldots, m$ define a *polynomial* (or *rational*) *map*

$$\widehat{g} \colon R_\infty \to R_\infty \quad \text{with} \quad \left(\widehat{g}(x)\right)_i = \begin{cases} \widehat{g}_i(x) & \text{if } 1 \leq i \leq m \\ x_i & \text{otherwise} \end{cases}$$

on R_∞. Note that $\widehat{g}_i \colon R_\infty \to R$ is a polynomial (or rational) map defined by g_i as in Definition 5.14.

Finally we must define input and output mappings allowing us to relate the input and output space R^∞ with the state space R_∞. As for the case of finite-dimensional machines, these mappings are associated with the input and output nodes.

Definition 5.16 ([Blu+98], p.71)**.** Let $I_\infty \colon R^\infty \to R_\infty$ and $O_\infty \colon R_\infty \to R^\infty$ be the maps defined by

$$I_\infty(x) = (\ldots, 0, 0, 0, \widehat{n} \,\#\, x_1, x_2, \ldots, x_n, 0, 0, \ldots)$$

for $x \in R^n$, where $\widehat{n}$ is the sequence of n 1s if $n > 0$ and $\widehat{0} = 0$, and

$$O_\infty(\ldots, x_0 \,\#\, x_1, \ldots, x_l, \ldots) = \begin{cases} 0 \in R^0 & \text{if } l = 0 \\ (x_1, \ldots, x_l) & \text{otherwise} \end{cases}$$

where $l = \min_{i \geq 0}\{x_{-i} = 0\}$.

Definition 5.17 ([Blu+98], p. 71)**.** A *Blum-Shub-Smale (BSS) machine* M over a ring R is a finite connected graph, containing five types of nodes: input, computation, branch, output, and *shift* nodes. The Space R^∞ is both the underlying input and output space of M, and R_∞ is the state space. Input, computation, branch and output nodes are defined as in the case of finite-dimensional machines (5.2). The only difference being that now the notion of polynomial (or rational) maps on R_∞ are used and the linear input and output maps are substituted by I_∞ and O_∞, respectively. Each shift node has possibly multiple incoming edges and one outgoing edge. In addition, each shift node η has associated with it an endomorphism $g_\eta \in \{\sigma_L, \sigma_R\}$ of the state space into itself and a unique next node β_η.

By replacing the input and output spaces by R^∞ and the configuration space by R_∞, the definitions for the computing endomorphism, computation, computation path, halting computation, input-output maps, computable maps, decidable and undecidable sets are exactly as in the case of finite-dimensional machines with some adaptions [Blu+98, p. 72]. It is important to note, that the increased computational power of a machine as defined above is not a result of the infinite dimensional state space per se, but the shift nodes that enable the accessing of "registers of arbitrary high address" [Blu+98, p. 71]. In the case of $R = \mathbb{Z}_2$, this formulation reduces to the classical definition of Turing machine operating on a "tape" of Boolean values.

Definition 5.18 ([AB07], Definition 16.23)**.** The class $\mathsf{P}_\mathbb{R}$ contains every language over $\mathbb{R}$, i.e. subsets of $\mathbb{R}^\infty$, that are decidable by a BSS machine over $\mathbb{R}$ in polynomial time.

Definition 5.19 ([All+09], Section 1.1)**.** The set $\mathsf{P}_\mathbb{R}^S$ is the subset languages L in P_real, that are decidable in polynomial time by a BSS machine over $\mathbb{R}$ using only machine constants in S.

Note that $\mathsf{P}_\mathbb{R} = \bigcup_{S \subseteq \mathbb{R}} \mathsf{P}_\mathbb{R}^S$.

5.3 Straight-Line Programs and the Problem PosSLP

Definition 5.20 ([AB07], Definition 16.2)**.** A *straight-line program* (SLP) of size S is a sequence of instructions of the form $y_i := z_k \diamond z_l$ for $i = 1, \ldots, S$, where $\diamond$ is an operation of the underlying ring or field, i.e. $\diamond \in \{+, -, \times, \div\}$ in case of $\mathbb{R}$, and z_k and z_l are either input variables, built-in constants or intermediate values y_j for $j < i$. For every choice of input variables, executing these simple instructions in order yields values for $y_1, \ldots, y_S$. We call the value of y_S the *output* of the computation. We call a SLP *division-free* if it does not make use of the division operation.

If a division-free SLP P that uses only constants in $\{0, 1\}$ has no indeterminates, i.e. input variables, the result of P is an integer X. In this case we say that P *represents* X.

Example. Let n be some positive integer. Consider the following SLP:

$$\begin{aligned} y_0 &:= 2 \\ y_1 &:= y_0 \cdot y_0 \\ &\vdots \\ y_n &:= y_{n-1} \cdot y_{n-1} \end{aligned}$$

Its easy to show by induction, that this SLP outputs 2^{2^n} since y_i evaluates to 2^{2^i} for every $i \leq n$:

$$y_0 = 2 \quad \text{and} \quad y_i = y_{i-1} \cdot y_{i-1} = 2^{(2^{i-1}+2^{i-1})} = 2^{2^i}.$$

This example demonstrates that a SLP of size n can represent a double exponentially large integer. In contrast, representing this number in binary would require 2^n bits.

Lemma 5.21 ([Str73])**.** *Every SLP of size n can be transformed into an equivalent division-free SLP of size polynomial in n.*

We now introduce the problem PosSLP, whose complexity we study in the subsequent chapter.

Definition 5.22 ([All+09])**.** For every finite subset $S \subset \mathbb{R}$, PosSLP(S) is the language of division-free SLPs without indeterminates that use only constants in $S \cup \{0, 1\}$ and evaluate to a positive real number.

Definition 5.23. PosSLP is the language of SLPs that represent a positive integer.

In other words, a SLP P is in PosSLP if it is in PosSLP($\emptyset$).

5.4 Boolean Parts

Definition 5.24 ([Blu+98], Section 22.2, Definition 2)**.** Given a class $\mathcal{C}$ of subsets of $\mathbb{R}^\infty$ define the *Boolean part* of $\mathcal{C}$ as

$$\mathsf{BP}(\mathcal{C}) := \{S \cap \mathbb{Z}_2^\infty \mid S \in \mathcal{C}\},$$

where we are identifying $\mathbb{Z}_2$ with the subset $\{0, 1\}$ of $\mathbb{R}$.

In Chapter 7, we prove Theorem 1.3 by reducing SQRT-SUM to the problem PosSLP in two steps: First, we show how SQRT-SUM can be solved in polynomial time by a machine which is equivalent to the BSS machines presented in this chapter. The following theorem states that the set of problems over $\{0,1\}^*$ decided by a BSS machine over the reals using a finite set of real constants, i.e. $\mathsf{BP}(\mathsf{P}_{\mathbb{R}}^{S})$, can be decided by a standard polynomial-time TM having access to an oracle for the language PosSLP. We use this fact in the second step, to show that SQRT-SUM $\in \mathsf{P}^{\mathrm{PosSLP}}$. By Theorem 1.2, which we address in the following chapter, it then follows that PosSLP is in CH.

Theorem 5.25 ([All+09], Proposition 1.1)**.** $\mathsf{P}^{\mathrm{PosSLP}(S)} = \mathsf{BP}(\mathsf{P}_{\mathbb{R}}^{S})$ *for all finite subsets* $S \subset \mathbb{R}$*.*

Proof. Let P be a SLP of size n. Since a BSS-machine over $\mathbb{R}$ can evaluate every arithmetic expressions used in P in $\mathcal{O}(1)$, we can implement a SLP interpreter that evaluates the result of P in linear time and perform a single sign-test afterwards. Thus, a machine performing a polynomial number of queries to a PosSLP-oracle can be emulated by a BSS machine in polynomial time, i.e. $P^{\mathrm{PosSLP}(S)} \subseteq \mathsf{BP}(\mathsf{P}_{\mathbb{R}}^{S})$.

To show the other direction, let M be a polynomial-time BSS machine over $\mathbb{R}$ using only constants in $S \cup \{0,1\}$. According to Lemma 5.21 we may assume that M does not make use of division, without loss of generality. We can emulate M using a standard polynomial-time TM M' having access to a PosSLP(S)-oracle as follows: Given this input string M performs some computation and produces intermediate values which are stored in the cells of M. For every cell c_i of M, the standard TM M' stores a SLP P_i which initially represents the initial content of the cell, i.e. 0 or 1 since we are considering the Boolean part. Instead of computing and storing the actual values of these intermediates, M appends every arithmetic operation performed on cell c_i to P_i. Note that since M can access only polynomially many cells of its tape and can only perform a polynomial number of operations on each cell, M' must only store polynomially many SLPs of polynomial size. M can then simulate branching instructions of M', i.e. evaluate the branching function, by sending the constructed SLP to its PosSLP(S)-oracle. □

Corollary 5.26. $\mathsf{P}^{\mathrm{PosSLP}} = \mathsf{BP}(\mathsf{P}_{\mathbb{R}}^{\emptyset})$.

6. PosSLP is in the Counting Hierarchy

We prove that the problem PosSLP, introduced in Chapter 5, lies within the counting hierarchy CH. As mentioned in Chapter 1, we follow the proof of Allender et. al closely. However, the proof, more specifically the proof of Lemma 4.4. in [All+09], contains two arguments that are beyond our understanding. We present the statement and use it for the rest of our proof, however, we refrain from formally showing its correctness. Instead, we highlight the two critical arguments and prove the statement under assumption that they are true. Furthermore, we show a slightly weaker statement, which only depends on one of the assumptions and discuss the consequences.

In Chapter 4 we defined $\mathcal{P}_n$ as the set of all odd primes less than n and P_n as their product, respectively. Furthermore, recall that $N = N(n) = 2^{n^2}$; a value that becomes of importance in this chapter. P_N therefore denotes the product of all odd primes less than 2^{n^2}.

Definition 6.1. We define a family of approximation functions $app_n \colon \mathbb{R} \to \mathbb{R}$ as

$$app_n(X) = \sum_{p \in \mathcal{P}_n} x_p h_{p,n} \sigma_{p,n},$$

where σ_n is the result of truncating the binary expansion of $1/p$ after 2^{n^4} bits.

Lemma 6.2. *For sufficiently large n and $X < P_N$, it holds that $|app_n(X) - X/P_N| \leq 2^{-2^{n^3}}$.*

Proof. Using the triangle inequality and the fact that x_p and $h_{p,n}$ are elements of $\mathbb{Z}_p$ and consequently nonnegative, we can conclude:

$$\begin{aligned} |app_n(X) - X/P_N| &= \Big| \sum_{p \in \mathcal{P}_n} x_p h_{p,n} \sigma_{p,n} - \Big(\sum_{p \in \mathcal{P}_n} x_p h_{p,n} P_N/p \Big)/P_N \Big| \\ &= \Big| \sum_{p \in \mathcal{P}_n} x_p h_{p,n} (\sigma_{p,n} - 1/p) \Big| \\ &\leq \sum_{p \in \mathcal{P}_n} x_p h_{p,n} \, |\sigma_{p,n} - 1/p| \end{aligned}$$

The binary expansions of $1/p$ and $\sigma_{p,n}$ are identical up to the 2^{n^4}-th bit, thus

$$|\sigma_{p,n} - 1/p| \leq 2^{-2^{n^4}}.$$

Since p is always less than 2^{n^2}, $|\mathcal{P}_n| < 2^{n^2}$ and $x_p h_{p,n} \leq p^2 \leq 2^{2n^2}$. It follows that

$$\sum_{p \in \mathcal{P}_n} x_p h_{p,n} \left|\sigma_{p,n} - 1/p\right| \leq 2^{n^2} \cdot 2^{2n^2} \cdot 2^{-2^{n^4}} = 2^{-2^{n^4}+3n^2}.$$

It remains to show that $-2^{n^4} + 3n^2 \leq -2^{n^3}$ for large enough n, which follows directly from the fact that $2^{n^4} - 3n^2 \in \Omega(2^{n^3})$, i.e. $2^{n^4} - 3n^2$ grows at least as fast as 2^{n^3}. □

Theorem 6.3 ([All+09], Theorem 4.2)**.** PosSLP $\in \mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}}}$

Proof. Let P be a SLP of size n that represents the integer W. Denote as Y_n the number 2^{2^n}. Since $|W| \leq Y_n$ as a consequence of iterated squaring, the number $X = W + Y_n$ is nonnegative. We can construct a SLP of size $2n + 2$ that represents X using iterated squaring to compute Y_n in n steps and performing the addition with the result of P. The integer W is positive, i.e. $P \in$ PosSLP, if and only if $X > Y_n$. Owing to the fact that X and Y_n are integers, if $X > Y_n$, then X/P_N and Y/P_N differ by at least

$$1/P_N > 2^{-2^{n^2+1}},$$

where the inequality follows from Lemma 4.9. The result of the approximation function app_n is within $2^{-2^{n^3}}$ of the actual value which is smaller than $2^{-2^{n^2+1}}$ for large enough n. It is therefore sufficient to compare $app_n(X)$ and $app_n(Y_n)$ to decide whether $X > Y_n$.

If $app_n(X) > app_n(Y)$, there exists an integer j with $0 \leq j \leq 2^{n^4}$ such that all bits in the binary expansion of $app_n(X)$ and $app_n(Y)$ before the j-th bit are equal, but the j-th bit of $app_n(X)$ is 1 and the j-th bit of $app_n(Y)$ is 0. More formally we can write this condition as follows:

$$\exists j \leq 2^{n^4}\ \forall i < j : \big(app_n(X)_i = app_n(Y)_i\big) \wedge \big(app_n(X)_j = 1 \wedge app_n(Y)_j = 0\big).$$

Note that every nonnegative integer smaller than 2^{n^4} can be represented using $\log_2(2^{n^4}) = n^4$ bits. Instead of quantifying over the integers we can therefore quantify over bit strings of length n^4. Let us denote the binary representation of i and j as I and J, respectively.

We define A to be the language of tuples $(P, I, b, 1^n)$, where the bit at position I of the binary expansion of $app_n(X)$ is b, X is the number represented by the SLP P and I is given in binary. Let M be a TM having access to an A-oracle. When given an input (P_X, P_{Y_n}, J, I), the machine M accepts if and only if

- the i-th bit of X and Y_n are equal for $i < j$, i.e.

$$i < j \Rightarrow \big(A(P_X, I, 1, 1^n) \wedge A(P_{Y_n}, I, 1, 1^n)\big) \vee \big(A(P_X, I, 0, 1^n) \wedge A(P_{Y_n}, I, 0, 1^n)\big),$$

- and the j-th bit of X is 1, whereas the j-th bit of Y_n is 0, i.e.

$$A(P_X, J, 1, 1^n) \wedge A(P_{Y_n}, J, 0, 1^n).$$

We see that $X > Y_n$ if and only if

$$\exists J \in \{0,1\}^{n^4}\ \forall I \in \{0,1\}^{n^4} : M(P_X, P_{Y_n}, J, I) = 1.$$

Thus, according to Definition 2.6, whether $X > Y_n$ and therefore the language PosSLP, can be decided in $(\Sigma_2^p)^A \subseteq \mathsf{PH}^A$. We show that $A \in \mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}}}$ in Lemma 6.4. The claim, PosSLP $\in \mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}}}$, then follows from Lemma 2.8 since $\mathsf{PH}^{\mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}}}} = \mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}}}$. □

Lemma 6.4 ([All+09], Lemma 4.3)**.** $A \in \mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}}}$

Proof. A tuple $(P, j, b, 1^n)$ is in A if and only if the j-th bit of the binary expansion of $app(X)$ is b, where X is the integer represented by the SLP P and j is given in binary. Let B_p denote the number $x_p h_{p,n} \sigma_{p,n}$. Because of the factor $\sigma_{p,n}$ it suffices to compute the first 2^{n^4} bits of the binary expansion of the sum of the numbers B_p in order to decide A. Let B be the language of tuples $(P, j, b, p, 1^n)$, where the j-th bit of the number B_p is b, $p < 2^{n^2}$ is an odd prime, X is the number represented by the SLP P and j is given in binary. We can construct an algorithm that fetches the bits of B_p for all odd primes $p < 2^{n^2}$ by querying a B-oracle, and then computes the iterated sum of all numbers B_p. The problem of adding at most 2^{n^2} number having 2^{n^4} bits each is an instance of the problem ITADD (Problem 3.24). According to Lemma 3.27, the language BitITADD is in $\mathsf{PH}^{\mathsf{PP}}$. Thus, determining the value of a single bit of the sum $\sum_{p \in \mathcal{P}_n} x_p h_{p,n} \sigma_{p,n}$, i.e. deciding the language A, is in $\mathsf{PH}^{\mathsf{PP}^B}$.

Lemma 6.13 states that $B \in \mathsf{PH}^{\mathsf{PP}}$, thus

$$A \in \mathsf{PH}^{\mathsf{PP}^{\mathsf{PH}^{\mathsf{PP}}}}. \tag{6.1}$$

Corollary 2.15 states that $\mathsf{PP}^{\mathsf{PH}^O} \subseteq \mathsf{P}^{\mathsf{PP}^O}$ for every oracle O, hence

$$\mathsf{PP}^{\mathsf{PH}^{\mathsf{PP}}} \subseteq \mathsf{P}^{\mathsf{PP}^{\mathsf{PP}}}. \tag{6.2}$$

In conclusion:

$$A \in \mathsf{PH}^{\mathsf{PP}^{\mathsf{PH}^{\mathsf{PP}}}} \subseteq \mathsf{PH}^{\mathsf{P}^{\mathsf{PP}^{\mathsf{PP}}}} = \mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}}},$$

where the inclusion follows from inserting 6.2 into 6.1 and the equation follows since $\mathsf{PH}^{\mathsf{P}} = \mathsf{PH}$ by Corollary 2.9. □

Definition 6.5. We denote as C the language of tuples (q, g, i, p), where p is prime with $p \neq q$ and i is the least number for which $g^i \equiv q \bmod p$.

Lemma 6.6. $C \in \mathsf{PH}$

Proof. We can test whether q and p are prime in polynomial time using the Agrawal–Kayal–Saxena primality test [AKS04]. The condition that i is the least number that satisfies $g^i \equiv q \bmod p$ can be written more formally as

$$\forall j < i : g^i \equiv q \bmod p \wedge g_j \not\equiv q \bmod p.$$

The number i may be exponentially large but its binary representation still has polynomially many bits. Instead of quantifying over integers, we can therefore instead quantify over polynomial sized bit strings. Therefore,

$$x := (q, g, i, p) \in C \iff \forall J \in \{0,1\}^{p(|x|)} \colon (j < i) \to (g^i \equiv q \bmod p \wedge g_j \not\equiv q \bmod p),$$

where p is a polynomial and j is the number represented by J. According to Definition 2.6 $C \in \Pi_1^p$, which is a subset of PH, if the matrix of the above formula can be decided by a polynomial time TM. We can obviously test whether $j < i$ in polynomial time by comparing their binary representations. It remains to show that we can also decide the consequence of the implication in polynomial time.

Note i may be an exponentially large number in comparison to the length of the binary representation of i. The naïve way of computing g^i needs i multiplications and is therefore

not feasible. To conclude the proof we introduce the so-called *square-and-multiply* method which solves this problem in time logarithmic in the exponent, thus in polynomial time.

By the rules of modular arithmetic, $ab \bmod p = (a \bmod p)(b \bmod p) \bmod p$ and therefore $a^b \bmod p = (a \bmod p)^b \bmod p$ for any $a, b \in \mathbb{Z}$. Thus, we can compute $g^i \bmod p$ using the following recursive formula:

$$g^i \bmod p = \begin{cases} 1 & \text{if } i = 0 \\ (g^2 \bmod p)^{i/2} \bmod p & \text{if } i \text{ is even} \\ (g \bmod p)(g^2 \bmod p)^{(i-1)/2} \bmod p & \text{if } i \text{ is odd.} \end{cases}$$

Note that all remainders in the above formula can be computed in polynomial time. This method needs $\mathcal{O}(\log_2 i)$ iterations hence $g^i \bmod p$ is computable in polynomial time. □

Definition 6.7. The function $\mathrm{dlog}\colon \mathbb{N}^3 \to \mathbb{N}$ maps a tuple (q, g_p, p) to the *discrete logarithm* of q to a base g_p modulo a prime p, i.e. the smallest number $i \in \mathbb{N}$ such that $g_p^i \equiv q \bmod p$.

Lemma 6.8. $\mathrm{dlog} \in \#\mathsf{P}^C$

Proof. Let M be a nondeterministic TM. Given numbers q, p and g_p as input, M non-deterministically guesses a number i and performs a call to the oracle C in order to test if $(q, g_p, i, p) \in C$. If the tuple (q, g_p, i, p) is an element of C, i is the least number for which $g_p^i \equiv q \bmod p$ and thus i is the discrete logarithm of X to the base g_p modulo p. In this case M generates i accepting paths. One way to achieve this is by non-deterministically guessing a number $j \in \mathbb{Z}/i\mathbb{Z}$ and accepting each result. By the Definition 2.10, a function f is in $\#\mathsf{P}$ if $f(x)$ is equal to the number of accepting path of a nondeterministic polynomial-time TM for every input $x \in \{0,1\}^*$. □

Definition 6.9. BitDLOG is the language of tuples (q, g_p, p, j), where j is given in binary, such that the j-th bit of the binary expansion of $\mathrm{dlog}(q, g_p, p)$ is 1.

Corollary 6.10. $\mathrm{BitDLOG} \in \mathsf{P}^{\mathsf{PP}}$

Proof. Despite the fact that the value of j might be exponentially large in the size of the input, the value of $\mathrm{dlog}(q, g_p, p)$ can be represented by a polynomial number of bits. Querying bits at indices higher than the length of $\mathrm{dlog}(q, g_p, p)$ does not make much sense. Therefore, we can assume that j is polynomial in the size of the input.

A TM having access to a dlog-oracle can decide BitDLOG by simply querying the result of $\mathrm{dlog}(q, g_p, p)$ and testing whether the j-th bit of the result is 1. Since the oracle gives answers in constant time and j is polynomial in the size of the input, as noted above, this is achieved in polynomial time.

Combining the results of the two preceding lemmas (6.6 and 6.8), we see that $\mathrm{dlog} \in \#\mathsf{P}^{\mathsf{PH}}$, implying that

$$\mathrm{BitDLOG} \in \mathsf{P}^{\#\mathsf{P}^{\mathsf{PH}}}.$$

Lemma 2.12 states that $\mathsf{P}^{\#\mathsf{P}} = \mathsf{P}^{\mathsf{PP}}$. Therefore, $\mathsf{P}^{\#\mathsf{P}^{\mathsf{PH}}} = \mathsf{P}^{\mathsf{PP}^{\mathsf{PH}}}$ and $\mathsf{P}^{\mathsf{PP}^{\mathsf{PH}}} \subseteq \mathsf{P}^{\mathsf{P}^{\mathsf{PP}}} = \mathsf{P}^{\mathsf{PP}}$, by Corollary 2.15 and Lemma 2.5. In summary the following equivalences and inclusions hold:

$$\mathrm{BitDLOG} \in \mathsf{P}^{\mathrm{dlog}} \subseteq \mathsf{P}^{\#\mathsf{P}^C} \subseteq \mathsf{P}^{\#\mathsf{P}^{\mathsf{PH}}} = \mathsf{P}^{\mathsf{PP}^{\mathsf{PH}}} \subseteq \mathsf{P}^{\mathsf{P}^{\mathsf{PP}}} = \mathsf{P}^{\mathsf{PP}}.$$ □

Remark. Allender et al. state that the discrete logarithm of a number to a base g_p modulo p can be computed in P^{PP} since $\#\mathsf{P}^C \subseteq \mathsf{P}^{\mathsf{PP}}$. In our understanding, this inclusion cannot be formally correct because $\#\mathsf{P}$ is a set of counting problems, that is functions mapping problem instances to nonnegative integers, and P is a set of decision problems. For this reason, we differentiate between the function dlog and the language BitDLOG.

The following lemma shows that the problem of deciding whether an element of the integers modulo a prime p a generator of this group and in particular the *least* generator lies within the polynomial hierarchy. As all finite fields, $\mathbb{Z}/p\mathbb{Z}$ has no order. By saying that an element $g \in \mathbb{Z}/p\mathbb{Z}$ is the *least* generator we mean that g is less than all other generators when viewing $\mathbb{Z}/p\mathbb{Z}$ as a subset of $\mathbb{Z}$. This is merely a technical detail.

Lemma 6.11. *The language* LG *of tuples* (g, p) *such that* p *is a prime and* g *is the least generator of the multiplicative group of the integers modulo* p *is in* PH.

Proof. To simplify notation, we omit writing *modulo* p in all arithmetic expressions. Recall the definition of a group generator: An element g of a group G of order m is a generator of G if it has order m. Let $\mathrm{Gen}(g)$ denote the following first-order formula that is satisfied if and only if $g \in G$ is a generator:

$$\forall i\colon i < m \to (g^i \not\equiv 1 \land g^m = 1).$$

Then g is the least generator if every element less than g is not a generator, i.e.

$$\forall g'\colon g' < g \to (\neg\mathrm{Gen}(g') \land \mathrm{Gen}(g)).$$

We can test whether p is prime in polynomial time using the Agrawal–Kayal–Saxena primality test [AKS04]. To show that deciding whether g is the least generator of $(\mathbb{Z}/p\mathbb{Z})^\times$ is in PH, we transform the above first-order formula into *prenex normal form* with alternating quantifiers, that is all quantifiers appear at the front of the formula followed by a quantifier free matrix and all quantifiers are alternating. In addition we show that the matrix can be evaluated by a polynomial time TM and all quantified variables can be represented using bit-strings of polynomial length.

Inserting the formulas for $\mathrm{Gen}(g')$ and $\mathrm{Gen}(g)$ yields

$$\forall g'\colon g' < g \to \Big[\neg(\forall i\colon i < m \to (g'^i \neq 1 \land g'^m = 1)) \land (\forall i\colon i < m \to (g^i \neq 1 \land g^m = 1))\Big].$$

For every formula ϕ, $\neg(\exists x\colon \phi(x))$ is equivalent to $\forall x\colon \neg\Phi(x)$. After moving the negation into the matrix of $\mathrm{Gen}(g')$ like above and applying De Morgan's law we are left with

$$\forall g'\colon g' < g \to \Big[(\exists i\colon i < m \land (g'^i = 1 \lor g'^m \neq 1)) \land (\forall i\colon i < m \to (g^i \neq 1 \land g^m = 1))\Big].$$

Finally, after renaming the variable i in the second formula, we can move all quantifiers to the front, yielding

$$\forall g'\ \exists i\ \forall j\colon g' < g \to \Big[i < m \land (g'^i = 1 \lor g'^m \neq 1) \land (j < m \to g^j \neq 1 \land g^m = 1)\Big].$$

It remains to show that the now quantifier-free matrix can be evaluated in polynomial time. The powers of g' and g can be computed using the square-and-multiply method presented in Lemma 6.6. Evaluating the inequalities and logical connectives can also be achieved in polynomial time. Furthermore, all quantified variables can be represented using bit-strings of polynomial size. By Definition 2.6 the language LG is therefore in $\Pi_3^p \subseteq \mathsf{PH}$. □

Lemma 6.12. *Given a SLP of size n representing an integer X and a prime p, we can compute x_p, i.e. X mod p, in time polynomial in p and n.*

Proof. Let P be a SLP of size n that represents an integer X, i.e. the last instruction y_n evaluates to X. Thus, $x_p = y_n \bmod p$. Since P represents an integer, it consists only of instruction of the form:

$$y_i := z_k + z_l \quad \text{and} \quad y_i := z_k \times z_l,$$

where z_k and z_l are either intermediate values y_j for $j < i$ or constants in $\{0, 1\}$. By the rules of modular arithmetic, $a + b \bmod c = ((a \bmod c) + (b \bmod c)) \bmod c$ and $ab \bmod c = ((a \bmod c)(b \bmod c)) \bmod c$, for all integers a, b, c. Therefore,

$$y_i \bmod p = z_k \diamond z_l \bmod p = ((z_k \bmod p) \diamond (z_l \bmod p)) \bmod p,$$

for all $i \leq n$, where $\diamond$ is either $+$ or $\times$. Note, that if we perform the modulo operation in each step, non of the intermediate values exceeds p. Hence, each step can be performed in time polynomial in p. Since P has exactly n instructions, computing x_p can be accomplished in time polynomial in p and n. □

Lemma 6.13 ([All+09], Lemma 4.4)**.** $B \in \mathsf{PH}^{\mathsf{PP}}$.

We are not able to fully validate all steps in the proof of this statement, as given by Allender et al. Instead, we define two assumptions and proof Lemma 6.13 relative to these assumptions. Furthermore, we show the slightly weaker statement $B \in \mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}}}$ which only depends on one of them.

Assumption 1. *The number $P_N/p \bmod p$ can be obtained in P^{PP}.*

Assumption 2. *Computing the product $x_p h_{p,n} \sigma_{p,n}$ is an instance of the problem* LOGITADD*, that is the product can be computed as the sum of $\log n$ many n-bit numbers.*

Lemma 6.14. *$B \in \mathsf{PH}^{\mathsf{PP}}$ relative to Assumptions 1 and 2.*

Proof. A tuple $(P, j, b, p, 1^n)$ is in B if and only if j is given in binary and the following conditions are satisfied:

1. p is an odd prime and less than 2^{n^2} and
2. the j-th bit of the number $B_p = x_p h_{p,n} \sigma_{p,n}$ is b,

where X is the integer represented by the SLP P and $x_p = X \bmod p$. We show that the first condition can be tested in P and the second condition can be tested in $\mathsf{PH}^{\mathsf{PP}}$. Since $\mathsf{P} \subseteq \mathsf{PH} \subseteq \mathsf{PH}^{\mathsf{PP}}$, this implies that $B \in \mathsf{PH}^{\mathsf{PP}}$.

We can test whether p is prime in polynomial time using the Agrawal–Kayal–Saxena primality test [AKS04]. Furthermore, whether $p < 2^{n^2}$ can be tested by checking if the length of p without leading zeros is less than n^2. Hence, the first condition can be decided in P.

The second condition needs more work. We use findings from circuit complexity to show that the product $x_p h_{p,n} \sigma_{p,n}$ is computable in $\mathsf{PH}^{\mathsf{PP}}$. But first, we show how the factors x_p, $\sigma_{p,n}$ can be obtained in $\mathsf{P} \subseteq \mathsf{P}^{\mathsf{PP}}$, and $h_{p,n}$ can be computed in P^{PP} under Assumption 1:

- Lemma 6.12 shows how to compute x_p in polynomial time. Since $\mathsf{P} \subseteq \mathsf{P}^{\mathsf{PP}}$, x_p can be obtained in P^{PP}.

- By Lemma 4.20, the j-th bit of $1/p$ is 1 if and only if $2^j \bmod p$ is odd. To compute the bits of $\sigma_{p,n}$ we have to evaluate $2^j \bmod p$ and check if the least significant bit is 1. However, the number j may be exponentially large because the only requirement is that the binary expansion of j has polynomial length. Hence, a naïve algorithm has to perform exponentially many multiplications. The square-and-multiply method presented in Lemma 6.6 provides remedy. In conclusion, computing $\sigma_{p,n}$ is in P and, as already stated, $\mathsf{P} \subseteq \mathsf{P}^{\mathsf{PP}}$.
- The multiplicative group of the integers modulo p is cyclic, as shown in Theorem 4.19. Hence, it is generated by a single element. We can determine the least generator g_p of $(\mathbb{Z}/p\mathbb{Z})^\times$ in PH using nondeterminism to guess g_p and verifying that guess as described in Lemma 6.11. PH is a subset of P^{PP} according to Toda's Theorem (2.13).

  Each bit of the discrete logarithm of a number to the basis g_p is computable in P^{PP} by Corollary 6.10. Allender et al. claim without proof that the number $P_N/p \bmod p$ is obtained in P^{PP} by first computing the discrete logarithm r of $P_N/p \bmod p$ and then evaluating $g_p^r \bmod p$. We are not able to verify their argument, since it is unclear for us how this is accomplished without nondeterministically guessing the value $P_N/p \bmod p$. In our understanding, this requires some way to verify said guess in polynomial time but P_N/p is a double-exponentially large number.

  According to Corollary 4.17 the modular inverse of any number $a \in \mathbb{Z}/p\mathbb{Z}$ is given by $a^{p-2} \bmod p$. Therefore, $h_{p,n} = (P_N/p \bmod p)^{p-2} \bmod p$, which can be computed in polynomial time from $P_N/p \bmod p$. Under Assumption 1, $P_N/p \bmod p$ can be obtained in P^{PP}, hence computing $h_{p,n}$ is in P^{PP}.

Under Assumption 2, the product $x_p h_{p,n} \sigma_{p,n}$ is an instance of LOGITADD and therefore computable by a uniform AC^0 circuit according to Lemma 3.29. By Lemma 3.30, the value of the j-th output of any AC^0 circuit can be computed in PH. Therefore, the complete procedure of determining the factors x_p, $h_{p,n}$ and $\sigma_{p,n}$ and then computing their product is in $\mathsf{PH}^{\mathsf{P}^{\mathsf{PP}}}$. Finally, $\mathsf{PH}^{\mathsf{P}} = \mathsf{PH}$ by Corollary 2.9 and therefore $B \in \mathsf{PH}^{\mathsf{PP}}$. □

Remark. LOGITADD is the problem of computing the sum of n binary numbers of $\log n$ bits each (see Problem 3.28). Allender et al. state without proof that LOGITADD can be applied to compute the product $x_p h_{p,n} \sigma_{p,n}$. The number $\sigma_{p,n}$ has a length of 2^{n^4} bits. Both x_p and $h_{p,n}$ are less than p and p is less than 2^{n^2}. Hence, the product $x_p h_{p,n}$ is less than 2^{2n^2}. Computing the product $x_p h_{p,n} \sigma_{p,n}$ is equivalent to adding $\sigma_{p,n}$ to itself $x_p h_{p,n}$ times, i.e. the sum of 2^{2n^2} many 2^{n^4} bit numbers. However, $\log 2^{n^4} = n^4 < 2^{2n^2}$ for all $n \in \mathbb{N}$. In our understanding, the number of summands, i.e. 2^{2n^2}, is therefore too large for this sum to be an instance of LOGITADD. Computing the product, by adding $x_p h_{p,n}$ to itself $\sigma_{p,n}$ many times or any other combination, does not fix this problem either.

We are now able to show an important step towards the main theorem of this text.

Proof of Theorem 1.2. By Theorem 6.3, PosSLP $\in \mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}}}$ and Toda's Theorem (2.13) states that $\mathsf{PH} \subseteq \mathsf{P}^{\mathsf{PP}}$. Therefore,

$$\textsc{PosSLP} \in \mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}}} \subseteq \mathsf{P}^{\mathsf{PP}^{\mathsf{PP}^{\mathsf{PP}}}}.$$ □

6.1 Dropping one Assumption

Besides LOGITADD, we also defined the problem ITADD in Chapter 3 (see Problem 3.24). The task for ITADD is to compute the sum of n n-bit numbers. As explained above, we

are unable to validate that LOGITADD can be applied to compute the product $x_p h_{p,n} \sigma_{p,n}$. However, we show that the computation of said product is an instance of ITADD. This way, we are able to derive the following slightly weaker upper bound which depends only on Assumption 1:

Lemma 6.15. $B \in \mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}}}$ *relative to Assumption 1.*

Proof. Computing the product $x_p h_{p,n} \sigma_{p,n}$ is an instance of the problem ITADD: For $n > 1$, size of each summand $\sigma_{p,n}$ is 2^{n^4} and therefore at least as large as $x_p h_{p,n} = 2^{2n^2}$, which is the number of summands. Hence, the product is the sum of at most 2^{n^4} numbers, each having a length of 2^{n^4} bits. We show in the proof of Lemma 6.14 how factors x_p, $h_{p,n}$ and $\sigma_{p,n}$ can be obtained in P^{PP} under Assumption 1. By Lemma 3.27 the language BitITADD is in $\mathsf{PH}^{\mathsf{PP}}$. Therefore,

$$B \in \mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}}} \subseteq \mathsf{PH}^{\mathsf{PP}^{\mathsf{PH}^{\mathsf{PP}}}} \subseteq \mathsf{PH}^{\mathsf{P}^{\mathsf{PP}^{\mathsf{PP}}}} = \mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}}},$$

where the inclusions follow since $\mathsf{P} \subseteq \mathsf{PH}$, $\mathsf{PP}^{\mathsf{PH}} \subseteq \mathsf{P}^{\mathsf{PP}}$ by Theorem 2.14, and $\mathsf{PH}^{\mathsf{P}} = \mathsf{PH}$ by Corollary 2.9. □

We end the chapter by discussing how it would affect the complexity of PosSLP if Lemma 6.15 is true but Lemma 6.14 is not, i.e.

$$B \in \mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}}} \quad \text{but} \quad B \notin \mathsf{PH}^{\mathsf{PP}}.$$

Theorem 6.16. $B \in \mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}}} \implies \text{PosSLP} \in \mathsf{P}^{\mathsf{PP}^{\mathsf{PP}^{\mathsf{PP}^{\mathsf{PP}}}}}$

Proof. We show in the proof of Lemma 6.4 that $A \in \mathsf{PH}^{\mathsf{PP}^B}$. Hence, in this case

$$A \in \mathsf{PH}^{\mathsf{PP}^{\mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}}}}}.$$

By Corollary 2.15, $\mathsf{PP}^{\mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}}}} \subseteq \mathsf{P}^{\mathsf{PP}^{\mathsf{PP}^{\mathsf{PP}}}}$. Thus, $A \in \mathsf{PH}^{\mathsf{P}^{\mathsf{PP}^{\mathsf{PP}^{\mathsf{PP}}}}} \subseteq \mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}^{\mathsf{PP}}}}$. We show in the proof of Theorem 6.3 that PosSLP $\in \mathsf{PH}^A$. Therefore,

$$\text{PosSLP} \in \mathsf{PH}^{\mathsf{PP}^{\mathsf{PP}^{\mathsf{PP}}}}.$$

Toda's Theorem states that $\mathsf{PH} \subseteq \mathsf{P}^{\mathsf{PP}}$, thus

$$\text{PosSLP} \in \mathsf{P}^{\mathsf{PP}^{\mathsf{PP}^{\mathsf{PP}^{\mathsf{PP}}}}}.$$ □

In summary, if we need one more PP-oracle to decide B, PosSLP is the fourth level of the counting hierarchy instead of the third.

7. SQRT-SUM is in the Counting Hierarchy

We present an algorithm for approximating the square root of any nonnegative integer. This allows us to prove Theorem 1.2 (SQRT-SUM $\in \mathsf{CH}$) using the results from previous chapters.

7.1 Newton's Method

As taking square roots is not available as a primitive operation in the BSS model, we are interested in approximating this value up to arbitrary accuracy by evaluating a rational function. Computing the square root of some $a \in \mathbb{N}$ is achieved by approximating the positive root of the function $f(x) = x^2 - a$ using an algorithm known as *Newton's method*. We now describe Newton's method and take a closer look at using it for the problem of approximating square roots. More detailed expositions can be found in most introductory textbooks on numerical analysis, e.g. [Atk89].

Suppose that $x^* \in \mathbb{R}$ is the root of a function $f \in \mathcal{C}^2(\mathbb{R})$ and $x_n \in \mathbb{R}$ is an estimate of x^*. The Taylor series expansion around x_n is given by

$$f(x) = f(x_n) + f'(x_n)(x - x_n) + R(x)$$

for every $x \in \mathbb{R}$, with a remainder term $R(x) = \frac{1}{2}f''(c)(x - x_n)^2$ for some c between x and x_n. For $x = x^*$, the above formula yields

$$0 = f(x^*) = f(x_n) + f'(x_n)(x^* - x_n) + R(x^*).$$

Solving for x^* gives

$$x^* = x_n - \frac{f(x_n)}{f'(x_n)} - \frac{R(x^*)}{f'(x_n)}.$$

Let x_{n+1} be the value obtained from the above formula by assuming the remainder is zero, i.e.

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}.$$

Then the difference between the iterate x_{n+1} and the actual root x^* is given by

$$x^* - x_{n+1} = -R(x^*) = -\frac{1}{2}(x^* - x_n)\frac{f''(c)}{f'(x_n)}$$

for c between x^* and x_n. Using this formula, it it possible to show that Newton's method converges to x^* if the initial estimate x_0 is chosen sufficiently close to x^* and $f'(x^*) \neq 0$ [Atk89, Theorem 2.1]. This implies that the iteration $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$ can be used to approximate x^* up to the desired accuracy under the mentioned assumptions. The proof of convergence for the general case is out of the scope of this work. However, we show that Newton's method does in fact converge to $\sqrt{a}$ or $-\sqrt{a}$ for the function defined by $f(x) = x^2 - a$. In this case, the iteration formula is given by

$$x_{n+1} = x_n - \frac{x_n^2 - a}{2x_n} = \frac{1}{2}\Big(x_n + \frac{a}{x_n}\Big).$$

Theorem 7.1. *For every $x_0 \in \mathbb{R}$ and $a \in \mathbb{N}_0$, the absolute value of the sequence defined by the recursion $x_{n+1} = \frac{1}{2}(x_n + \frac{a}{x_n})$ converges to $\sqrt{a}$.*

Proof. Regardless of x_0, $|x_1| \geq \sqrt{a}$ because

$$\begin{aligned} x_1^2 - a &= \frac{1}{4}\Big(x_0^2 + 2a + \frac{a^2}{x_0^2}\Big) - a \\ &= \frac{1}{4}\Big(x_0^2 - 2a + \frac{a^2}{x_0^2}\Big) \\ &= \frac{1}{4}\Big(x_0 - \frac{a}{x_0}\Big)^2 \\ &= \frac{\big(x_0^2 - a\big)^2}{4x_0^2} \geq 0, \end{aligned}$$

where we use the definition of the recurrence in the first equality. Suppose $|x_n| \geq \sqrt{a}$ for $n \geq 1$. Then by the definition of the recurrence and the triangle inequality we see that

$$\begin{aligned} |x_{n+1}| - |x_n| &= \frac{1}{2}\Big|x_n + \frac{a}{x_n}\Big| - |x_n| \\ &\leq \frac{1}{2}\Big(|x_n| + \frac{a}{|x_n|}\Big) - |x_n| \\ &= \frac{a - |x_n|^2}{2|x_n|} \leq 0. \end{aligned}$$

Thus, for every $n \geq 1$, $|x_n|$ is decreasing and bound from below by $\sqrt{a}$. Hence, $|x_n|$ is converging and its limit $x = \lim_{n\to\infty} |x_n|$ satisfies $x = \Big|\frac{1}{2}\Big(x + \frac{a}{x}\Big)\Big|$. Since, as shown above, $x_n \geq 0$ for all $n \geq 1$, necessarily $x \geq 0$ and therefore

$$x = \frac{1}{2}\Big(x + \frac{a}{x}\Big) \iff x = \frac{a}{x} \iff x^2 = a \iff |x| = \sqrt{a} \iff x = \sqrt{a},$$

where the last equivalence also follows from $x \geq 0$. □

7.2 Reducing SQRT-SUM to PosSLP

Recall the definition of the decision problem SQRT-SUM: Given nonnegative integers $a_1, \ldots, a_n$ and $\delta_i \in \{-1, 1\}$ for $i = 1, \ldots, n$ decide whether

$$\sum_{i=1}^{n} \delta_i \sqrt{a_i} \geq 0.$$

Hence, the input for SQRT-SUM is a list of n nonnegative integers $a_1, \ldots, a_n$ and factors $\delta_1, \ldots, \delta_n$. Using Newton's Method, the result that PosSLP $\in \mathsf{CH}$, and the relationship between PosSLP and the Boolean part, we are now able to show our main theorem:

$$\text{SQRT-SUM} \in \mathsf{CH}.$$

Proof of Theorem 1.2. It was shown in [Tiw92] that SQRT-SUM can be decided on a *unit-cost algebraic random access machine* by approximating each square root using a polynomial number of Newton-iterations, multiplying each approximation of $\sqrt{a_i}$ by the coefficient $\delta_i \in \{-1, 1\}$ and adding all results. The proof that polynomially many iterations suffice involves results from algebraic number theory and is beyond the scope of this work. Unit-cost algebraic random access machines are to BSS-machines, like classical random access machines to classical TMs. In particular, unit-cost algebraic RAMs have the same computational power as BSS-machines but may offer polynomial speedup because the read-write head does not have to be moved to access a desired cell. Therefore, SQRT-SUM can be decided on a BSS-machine in polynomial time. The class of decision problems over the set $\{0,1\}^*$ decided by such machines is $\mathsf{BP}(\mathsf{P}^{\emptyset}_{\mathbb{R}})$. By Corollary 5.26, $\mathsf{P}^{\mathrm{PosSLP}} = \mathsf{BP}(\mathsf{P}^{\emptyset}_{\mathbb{R}})$, implying that SQRT-SUM $\in \mathsf{P}^{\mathrm{PosSLP}}$. Theorem 1.2 states that

$$\mathrm{PosSLP} \in \mathsf{P}^{\mathsf{PP}^{\mathsf{PP}^{\mathsf{PP}}}}$$

and in consequence

$$\mathrm{SQRT\text{-}SUM} \in \mathsf{P}^{\mathsf{P}^{\mathsf{PP}^{\mathsf{PP}^{\mathsf{PP}}}}} = \mathsf{P}^{\mathsf{PP}^{\mathsf{PP}^{\mathsf{PP}}}}$$

where the equality follows from Lemma 2.5. By Definition 2.16 SQRT-SUM lies within the third level of the counting hierarchy. □

Bibliography

[AB07] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge ; New York: Cambridge University Press, 2007. 579 pp. ISBN: 978-0-521-42426-4. DOI: 10.1017/CBO9780511804090.

[AE04] Dana Angluin and Sarah Eisenstat. *How Close Can $\sqrt{a} + \sqrt{b}$ Be to an Integer?* 2004. URL: https://www.cs.yale.edu/publications/techreports/tr1279.pdf.

[AKS04] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. "PRIMES Is in P". In: *Annals of Mathematics* 160.2 (Sept. 1, 2004), pp. 781–793. ISSN: 0003-486X. DOI: 10.4007/annals.2004.160.781.

[All+09] Eric Allender et al. "On the Complexity of Numerical Analysis". In: *SIAM Journal on Computing* 38.5 (Jan. 2009), pp. 1987–2006. ISSN: 0097-5397. DOI: 10.1137/070697926.

[Aro98] Sanjeev Arora. "Polynomial Time Approximation Schemes for Euclidean Traveling Salesman and Other Geometric Problems". In: *Journal of the ACM* 45.5 (Sept. 1998), pp. 753–782. ISSN: 0004-5411, 1557-735X. DOI: 10.1145/290179.290180.

[Atk89] Kendall E. Atkinson. *An Introduction to Numerical Analysis*. 2. ed., [14. print]. New York: Wiley, 1989. 693 pp. ISBN: 978-0-471-62489-9.

[AW97] Eric Allender and Klaus Wagner. *Counting Hierarchies: Polynomial Time And Constant Depth Circuits*. Nov. 30, 1997. URL: https://people.cs.rutgers.edu/~allender/papers/column40.pdf.

[BD23] Nikhil Balaji and Samir Datta. *USSR Is in P/Poly*. Nov. 1, 2023. DOI: 10.48550/arXiv.2310.19335. preprint.

[Blö91] Johannes Blömer. "Computing Sums of Radicals in Polynomial Time". In: [1991] Proceedings 32nd Annual Symposium of Foundations of Computer Science. IEEE Computer Society, Oct. 1, 1991, pp. 670–677. ISBN: 978-0-8186-2445-2. DOI: 10.1109/SFCS.1991.185434.

[Blu+98] Lenore Blum et al. *Complexity and Real Computation*. New York, NY: Springer New York, 1998. ISBN: 978-1-4612-6873-4 978-1-4612-0701-6. DOI: 10.1007/978-1-4612-0701-6.

[Blu04] Lenore Blum. "Computing over the Reals: Where Turing Meets Newton". In: *Notices of the AMS* 51.9 (2004), pp. 1024–1034. URL: https://www.ams.org/notices/200409/fea-blum.pdf.

[Bra+14] Marcus Brazil et al. "On the History of the Euclidean Steiner Tree Problem". In: *Archive for History of Exact Sciences* 68.3 (May 1, 2014), pp. 327–354. ISSN: 1432-0657. DOI: 10.1007/s00407-013-0127-z.

[Bur+00] C. Burnikel et al. "A Strong and Easily Computable Separation Bound for Arithmetic Expressions Involving Radicals". In: *Algorithmica* 27.1 (May 1, 2000), pp. 87–99. ISSN: 1432-0541. DOI: 10.1007/s004530010005.

[Can88] John Canny. "Some Algebraic and Geometric Computations in PSPACE". In: *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*. STOC '88. New York, NY, USA: Association for Computing Machinery, Jan. 1, 1988, pp. 460–467. ISBN: 978-0-89791-264-8. DOI: 10.1145/62212.62257.

[DMO09] Erik D. Demaine, Joseph S. B. Mitchell, and Joseph O'Rourke. *The Open Problems Project - Problem 33: Sum of Square Roots*. Sept. 9, 2009. URL: https://topp.openproblem.net/p33 (visited on 03/16/2024).

[EHS23] Friedrich Eisenbrand, Matthieu Haeberle, and Neta Singer. *An Improved Bound on Sums of Square Roots via the Subspace Theorem*. Dec. 4, 2023. URL: http://arxiv.org/abs/2312.02057. preprint.

[EY09] Kousha Etessami and Mihalis Yannakakis. "Recursive Markov Chains, Stochastic Grammars, and Monotone Systems of Nonlinear Equations". In: *Journal of the ACM* 56.1 (Feb. 3, 2009), 1:1–1:66. ISSN: 0004-5411. DOI: 10.1145/1462153.1462154.

[EY10] Kousha Etessami and Mihalis Yannakakis. "On the Complexity of Nash Equilibria and Other Fixed Points". In: *SIAM Journal on Computing* 39.6 (Jan. 2010), pp. 2531–2597. ISSN: 0097-5397. DOI: 10.1137/080720826.

[Fil15] Yuval Filmus. *Answer to "Proof of $P^{\#P} = P^{PP}$"*. Computer Science Stack Exchange. May 19, 2015. URL: https://cs.stackexchange.com/a/42733 (visited on 03/10/2024).

[For02] Lance Fortnow. *Complexity Class of the Week: PP*. Sept. 4, 2002. URL: https://blog.computationalcomplexity.org/2002/09/complexity-class-of-week-pp.html (visited on 02/05/2024).

[GGJ76] M. R. Garey, R. L. Graham, and D. S. Johnson. "Some NP-complete Geometric Problems". In: *Proceedings of the Eighth Annual ACM Symposium on Theory of Computing*. STOC '76. New York, NY, USA: Association for Computing Machinery, May 3, 1976, pp. 10–22. ISBN: 978-1-4503-7414-9. DOI: 10.1145/800113.803626.

[Haj+93] András Hajnal et al. "Threshold Circuits of Bounded Depth". In: *Journal of Computer and System Sciences* 46.2 (Apr. 1, 1993), pp. 129–154. ISSN: 0022-0000. DOI: 10.1016/0022-0000(93)90001-D.

[HAM02] William Hesse, Eric Allender, and David A. Mix Barrington. "Uniform Constant-Depth Threshold Circuits for Division and Iterated Multiplication". In: *Journal of Computer and System Sciences*. Special Issue on Complexity 2001 65.4 (Dec. 1, 2002), pp. 695–716. ISSN: 0022-0000. DOI: 10.1016/S0022-0000(02)00025-9.

[HPA12] John L. Hennessy, David A. Patterson, and Krste Asanović. *Computer Architecture: A Quantitative Approach*. 5. ed. Amsterdam Heidelberg: Elsevier, Morgan Kaufmann, 2012. ISBN: 978-0-12-383872-8 978-93-81269-22-0.

[HW79] Godfrey H. Hardy and Edward M. Wright. *An Introduction to the Theory of Numbers*. 5. ed. Oxford: Clarendon Press, 1979. 426 pp. ISBN: 978-0-19-853170-8 978-0-19-853171-5.

[IR90] Kenneth Ireland and Michael Rosen. *A Classical Introduction to Modern Number Theory*. Vol. 84. Graduate Texts in Mathematics. New York, NY: Springer, 1990. ISBN: 978-1-4419-3094-1 978-1-4757-2103-4. DOI: 10.1007/978-1-4757-2103-4.

[Kak87] S. C. Kak. "New Result on D-Sequences". In: *Electronics Letters* 23.12 (June 4, 1987), pp. 617–617. ISSN: 1350-911X. DOI: 10.1049/el:19870442.

[Mal01] Gregorio Malajovich. *An Effective Version of Kronecker's Theorem on Simultaneous Diophantine Approximation*. Oct. 5, 2001. URL: https://dma.im.ufrj.br/gregorio/papers/kron.pdf.

[Mit99] Joseph S. B. Mitchell. "Guillotine Subdivisions Approximate Polygonal Subdivisions: A Simple Polynomial-Time Approximation Scheme for Geometric TSP, k -MST, and Related Problems". In: *SIAM Journal on Computing* 28.4 (Jan. 1999), pp. 1298–1309. ISSN: 0097-5397, 1095-7111. DOI: 10.1137/S0097539796309764.

[MT98] Alexis Maciel and Denis Thérien. "Threshold Circuits of Small Majority-Depth". In: *Information and Computation* 146.1 (Oct. 10, 1998), pp. 55–83. ISSN: 0890-5401. DOI: 10.1006/inco.1998.2732.

[ORo81] J. O'Rourke. "Advanced Problem 6369". In: *The American Mathematical Monthly* 88.10 (1981), p. 769. DOI: 10.2307/2321488.

[Pap77] Christos H. Papadimitriou. "The Euclidean Travelling Salesman Problem Is NP-complete". In: *Theoretical Computer Science* 4.3 (June 1, 1977), pp. 237–244. ISSN: 0304-3975. DOI: 10.1016/0304-3975(77)90012-3.

[Par99] Ian Parberry. 1998 Gödel Prize. Mar. 25, 1999. URL: https://sigact.org/prizes/g%C3%B6del/1998.html (visited on 02/13/2024).

[PS88] Ian Parberry and Georg Schnitger. "Parallel Computation with Threshold Functions". In: *Journal of Computer and System Sciences* 36.3 (June 1, 1988), pp. 278–302. ISSN: 0022-0000. DOI: 10.1016/0022-0000(88)90030-X.

[Str73] Volker Strassen. "Vermeidung von Divisionen". In: *Journal für die reine und angewandte Mathematik* (1973), pp. 188–202. URL: https://gdz.sub.uni-goettingen.de/id/PPN243919689_0264.

[Tiw92] Prasoon Tiwari. "A Problem That Is Easier to Solve on the Unit-Cost Algebraic RAM". In: *Journal of Complexity* 8.4 (Dec. 1992), pp. 393–397. ISSN: 0885064X. DOI: 10.1016/0885-064X(92)90003-T.

[Tod91] Seinosuke Toda. "PP Is as Hard as the Polynomial-Time Hierarchy". In: *SIAM Journal on Computing* 20.5 (Oct. 1991), pp. 865–877. ISSN: 0097-5397. DOI: 10.1137/0220053.

[Vol99] Heribert Vollmer. *Introduction to Circuit Complexity: A Uniform Approach*. Texts in Theoretical Computer Science. Berlin ; New York: Springer, 1999. 270 pp. ISBN: 978-3-540-64310-4. DOI: 10.1007/978-3-662-03927-4.

[Wag86] Klaus W. Wagner. "The Complexity of Combinatorial Problems with Succinct Input Representation". In: *Acta Informatica* 23.3 (June 1, 1986), pp. 325–356. ISSN: 1432-0525. DOI: 10.1007/BF00289117.