Thesis check-in

A Thesis
Presented to
The Established Interdisciplinary Committee for
Mathematics and Computer Science
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Patrick Norton

November 26, 2024

Chapter 2

Relativization

An important prerequisite to understanding algebrization is the similar, but simpler, concept of *relativization*, also called *oracle separation*. To do this, we first must define an *oracle*.

Definition 2.0.1 ([2, Def. 2.1])**.** An *oracle* A is a collection of Boolean functions $A_m : \{0,1\}^m \to \{0,1\}$, one for each natural number m.

There are several ways to think of an oracle; this will extend the most naturally when it comes time to define an extension oracle later on. Another way to think of an oracle is as a subset $A \subseteq \{0,1\}^*$. This allows us to think of A as a language. Since we can do this, it gives us the ability to think of the complexity of the oracle. If we want to think about the subset in terms of our functions, we can write A as

$$A = \bigcup_{m\in\mathbb{N}} \{x \in \{0,1\}^m \mid A_m(x) = 1\}. \tag{2.1}$$

We will use the Iverson bracket defined earlier for this purpose: allowing us to think of A as the set and $[A]$ as the function.

Example 2.0.1.1. Let $m = 3$. The function

$$\begin{aligned} f : \{0,1\}^3 &\to \{0,1\} \\ abc &\mapsto b \end{aligned} \tag{2.2}$$

is an oracle function. We can think of f as corresponding to the set $\{010, 011, 110, 111\}$.

Example 2.0.1.2. For each $n \in \mathbb{N}$, define

$$\begin{aligned} f_n : \{0,1\}^n &\to \{0,1\} \\ a_1a_2\cdots a_n &\mapsto a_n. \end{aligned} \tag{2.3}$$

Then the set $\{f_n\}$ forms an oracle, whose corresponding language is the set of all binary representations of odd numbers.

An oracle is not particularly interesting mathematical object on its own (after all, it is simply a set of arbitrary Boolean functions); its utility comes from when it interacts with a Turing machine. A normal Turing machine does not have the facilities to interact with an oracle, so we need to define a small extension to a standard Turing machine to allow for this.

Definition 2.0.2 ([3, Def. 3.6])**.** A *Turing machine with an oracle* is a Turing machine with an additional tape, called the *oracle tape*, as well as three special states: q_{query}, q_{yes}, and q_{no}. Further, each machine is associated with an oracle A. During the execution of the machine, if it ever moves into the state q_{query}, the machine then (in one step) takes the output of A on the contents of the oracle tape, moving into q_{yes} if the answer is 1 and q_{no} if the answer is 0.

Of course, the question now becomes how we can effectively use an oracle in an algorithm. The previously-mentioned conception of an oracle as a set of strings is useful here. If we consider the set of strings as being a *language* in its own right, then querying the oracle is the same as determining whether a string is in the langauge, just in one step. If the language is computationally hard, this means our machine can get a significant power boost from the right oracle.

Definition 2.0.3 ([2, Def. 2.1])**.** For any complexity class $\mathcal{C}$, the complexity class $\mathcal{C}^A$ is the class of all languages determinable by a Turing machine with access to A in the number of steps defined for $\mathcal{C}$.

We will be using this definition in many places, so we should take a moment to look at it in more depth. First, it is important to realize that $\mathcal{C}^A$ is a set of *languages*, not *machines*: despite the notation, augmenting $\mathcal{C}$ with an oracle does not modify any languages, it just adds new ones that are computable. Second, since a machine can always ignore its oracle, it follows that adding an oracle can only increase the number of languages in the class, never decrease it.

Lemma 2.0.4. *For any complexity class $\mathcal{C}$ and oracle A, $\mathcal{C} \subseteq \mathcal{C}^A$.*

Proof. Let $L \in \mathcal{C}$ and M be a machine that determines L. Then the oracle machine M' that simulates M on its input and makes no queries to the oracle will also accept exactly L. Since M' is a $\mathcal{C}^A$ machine for any oracle A, it follows that $L \in \mathcal{C}^A$ and hence $\mathcal{C} \in \mathcal{C}^A$. □

While the above lemma tells us that $\mathcal{C} \subseteq \mathcal{C}^A$ always, another interesting question is when $\mathcal{C} = \mathcal{C}^A$. We do have a notion for this, called *lowness*. Lowness can be defined for both individual languages and complexity classes; we will define both here.

Definition 2.0.5. A language L is *low* for a class $\mathcal{C}$ if $\mathcal{C}^L = \mathcal{C}$.

Definition 2.0.6. A complexity class $\mathcal{D}$ is *low* for a class $\mathcal{C}$ if each language in $\mathcal{D}$ is low for $\mathcal{C}$.

Of particular interest to us will be classes that are low for *themselves*. We care about these classes because they can use other problems from the same class as a subroutine without issue; in particular recursion and iteration both work here. Thankfully, both P and PSPACE are low for themselves (it turns out NP is probably not); this allows us to easily write algorithms that recurse for classes in both of our most common classes.

Theorem 2.0.7. P *is low for itself.*

Proof. Let $L \in \mathsf{P}$ and let $K \in \mathsf{P}^L$. Let $M(L)$ be the determiner of L and $M(K)$ be the determiner of K. Further, let $\hat{M}(K)$ be the determiner of K but with access to L as an oracle. We aim to show $K \in \mathsf{P}$. Let $p_L(n)$ be a polynomial upper bound of the runtime of $M(L)$ on an input of length n, and let $p_{\hat{K}}(n)$ be similar. Since $M(K)$ can call $M(L)$ no more than $p_{\hat{K}}(n)$ times, it follows that $p_K(n) \leq p_{\hat{K}}(p_L(n))$. Hence, the runtime of $M(K)$ is bounded above by a polynomial, and thus $K \in P$. □

Theorem 2.0.8. PSPACE *is low for itself.*

Proof. The proof is very similar to that for Theorem 2.0.7, but with space instead of time. □

2.1 Defining relativization

We are now ready to define what relativization is. First, note that relativization is a statement about a *result*: we talk about inclusions relativizing, not sets themselves.

Definition 2.1.1. Let $\mathcal{C}$ and $\mathcal{D}$ be complexity classes such that $\mathcal{C} \subseteq \mathcal{D}$. We say the result $\mathcal{C} \subseteq \mathcal{D}$ *relativizes* if $\mathcal{C}^A \subseteq \mathcal{D}^A$ for all oracles A. Conversely, if there exists A such that $\mathcal{C} \nsubseteq \mathcal{D}$, we say that the result $\mathcal{C} \subseteq \mathcal{D}$ *does not relativize.*

Definition 2.1.2. Let $\mathcal{C}$ and $\mathcal{D}$ be complexity classes such that $\mathcal{C} \nsubseteq \mathcal{D}$. We say the result $\mathcal{C} \nsubseteq \mathcal{D}$ *relativizes* if $\mathcal{C}^A \nsubseteq \mathcal{D}^A$ for all oracles A. Conversely, if there exists A such that $\mathcal{C} \subseteq \mathcal{D}$, we say that the result $\mathcal{C} \nsubseteq \mathcal{D}$ *does not relativize.*

We start with a very straightforward example of a relativizing result.

Lemma 2.1.3. *For any oracle A, $\mathsf{P}^A \subseteq \mathsf{NP}^A$. Equivalently, the result $\mathsf{P} \subseteq \mathsf{NP}$ relativizes.*

Proof. Since any deterministic Turing machine is also a nondeterministic machine, it follows that a machine that solves a P^A problem is also an NP^A machine. Hence, $\mathsf{P}^A \subseteq \mathsf{NP}^A$. □

This result tells us that not *everything* is weird in the world of relativization (although we will soon do our best to find all the weird bits): if we have a machine that can do more operations without an oracle, it can still do more operations with an oracle. Further, for the question of P vs. NP that we will discuss in Section 2.3, this means that the question we care about is whether $\mathsf{NP} \subseteq^? \mathsf{P}$ relativizes. As such, the question we are asking simplifies to determining where $\mathsf{P}^A = \mathsf{NP}^A$ and where $\mathsf{P}^A \subsetneq \mathsf{NP}^A$.

Now that we have talked about set inclusions relativizing, we need to define the other side of the coin: *proofs* can relativize as well as results. Unfortunately, this needs to be a somewhat informal definition as formally delineating different types of proof is far beyond the scope of this paper. However, the definition we offer here will be sufficient for our purposes.

Definition 2.1.4. We say a *proof relativizes* if it is not made invalid if the relevant classes are replaced with oracle classes, i.e., a proof that $\mathcal{C} \subseteq \mathcal{D}$ *relativizes* if the same proof can be used to show $\mathcal{C}^A \subseteq \mathcal{D}^A$ for all oracles A with minimal modifications.

This gives us a reason to care about relativization as a concept: if our proofs are relativizing then we know not to try to use them to prove nonrelativizing results. In particular, we will show in Section 2.3 that the famous P vs. NP problem will not relativize regardless of the outcome, and then in Section 2.4 we will show that the common proof technique of diagonalization *does* in fact relativize.

Now that we have given ourselves a reason to care about oracles and how they interact with Turing machines, we now turn to the question of how a machine can gain information about the oracle it queries. We will do this with the notion of *query complexity.*

2.2 Query complexity

The goal of query complexity is to ask questions about some Boolean function $A : \{0,1\}^n \to \{0,1\}$ by querying A itself. For this, we will interchangeably think of A as a *function* as well as a bit string of length $N = 2^n$, where each string element is A applied to the ith string of length n, arranged in some lexicographical order. We can further think of the property itself as being a Boolean function; a function that takes as input the bit-string representation of A and outputs whether or not A has the given property. We will call the function representing the property f. When viewed like this, f is a function from $\{0,1\}^N$ to $\{0,1\}$. We define three types of query complexity for three of the most common types of computing paradigms: deterministic, randomized, and quantum. Nondeterministic query complexity is interesting, but it is outside the scope of this paper.

Definition 2.2.1 ([2, p. 17])**.** Let $f : \{0,1\}^N \to \{0,1\}$ be a Boolean function. Then the *deterministic query complexity* of f, which we write $D(f)$, is the minimum number of queries made by any deterministic algorithm with access to an oracle A that determines the value of $f(A)$.

To make this more clear, let us give an example problem.

Definition 2.2.2. The OR problem is the following oracle problem:

> Let $A : \{0,1\}^n \to \{0,1\}$ be an oracle. The function $\mathsf{OR}(A)$ returns 1 if there exists a string on which A returns 1, and 0 otherwise.

The question is then what the deterministic query complexity of the OR function is.

Theorem 2.2.3. *The* OR *problem has a deterministic query complexity of* 2^n*.*

Proof. First, note that any algorithm that determines the OR problem can stop as soon as it queries A and gets an output of 1. Hence, for any algorithm M, let $\{s_i\}$ be the sequence of queries M makes to A on the assumption that it always recieves a response of 0. If $|\{s_i\}| \leq 2^n$, there exists some $s \in \{0,1\}^n$ not queried. In that case, M will not be able to distinguish the zero oracle from the oracle that outputs 1 only when given s. Hence, M must query every string of length n and thus the query complexity is 2^n. □

From this, we get that the OR problem cannot be solved any better than by enumerative checking. This makes intuitive sense because none of the results we get by querying A imply anything about what A will do on other values, since A can be an arbitrary function. Later on (in **??**), we will look at what happens when we give ourselves access to a *polynomial*, where querying one point could tell us information about others.

For the next two definitions, since their Turing machines include some element of randomness, we only require that they succeed with a 2/3 probability. This is in line with most definitions of complexity classes involving random computers.

Definition 2.2.4 ([2, p. 17])**.** Let $f : \{0,1\}^N \to \{0,1\}$ be a Boolean function. Then the *randomized query complexity* of f, which we write $D(f)$, is the minimum number of queries made by any randomized algorithm with access to an oracle A that evaluates $f(A)$ with probability at least 2/3.

Definition 2.2.5 ([2, p. 17])**.** Let $f : \{0,1\}^N \to \{0,1\}$ be a Boolean function. Then the *quantum query complexity* of f, which we write $D(f)$, is the minimum number of queries made by any quantum algorithm with access to an oracle A that evaluates $f(A)$ with probability at least 2/3.

2.3 Relativization of P vs. NP

An important example of relativization is that of P and NP. While the question of if $\mathsf{P} = \mathsf{NP}$ is still open, we aim to show that *regardless of the answer*, the result does not algebrize. To do this, we show that there are some oracles A where $\mathsf{P}^A = \mathsf{NP}^A$, and some where $\mathsf{P}^A \neq \mathsf{NP}^A$.

Additionally, it should be noted that the similarity of relativization to algebrization means that the structure of these proofs will return in **??** when we show the algebrization of P and NP.

2.3.1 Equality

The more straightforward of the two proofs is the oracle where $\mathsf{P}^A = \mathsf{NP}^A$, so we shall begin with that.

Theorem 2.3.1 ([5, Theorem 2])**.** *There exists an oracle A such that* $\mathsf{P}^A = \mathsf{NP}^A$*.*

Proof. For this, we can let A be any PSPACE-complete language. By letting our machine in P be the reducer from A to any other language in PSPACE, we therefore get that $\mathsf{PSPACE} \subseteq \mathsf{P}^A$. Similarly, if we have a problem in NP^A, we can verify it in polynomial space without talking to A at all (by having our machine include a determiner for A). Hence, we have that $\mathsf{NP}^A \subseteq \mathsf{NPSPACE}$. Further, a celebrated result of Savitch [19] (which we briefly discussed as **??**) is that $\mathsf{PSPACE} = \mathsf{NPSPACE}$. Combining all these results, we get the chain

$$\mathsf{NP}^A \subseteq \mathsf{NPSPACE} = \mathsf{PSPACE} \subseteq \mathsf{P}^A \subseteq \mathsf{NP}^A. \tag{2.4}$$

This is a circular chain of subset relations, which means everything in the chain must be equal. Hence, $\mathsf{P}^A = \mathsf{NP}^A = \mathsf{PSPACE}$. □

For a slightly more intuitive view of what this proof is doing, what we have done is found an oracle that is so powerful that it dwarfs any amount of computation our actual Turing machine can do. Hence, the power of our machine is really just the same as the power of our oracle, and since we have given both the P and NP machine the same oracle, they have the same power.

2.3.2 Inequality

Having shown that an oracle exists where $\mathsf{P}^A = \mathsf{NP}^A$, we now endeavor to find one where $\mathsf{P}^A \neq \mathsf{NP}^A$. This piece of the proof is less simple than the previous section, and it uses a diagonalization argument to construct the oracle. Before we dive in to the main proof, however, we need to define a few preliminaries.

Definition 2.3.2 ([5, p. 436])**.** Let X be an oracle. The language $L(X)$ is the set

$$L(X) = \{x \mid \text{there is } y \in X \text{ such that } |y| = |x|\}.$$

Example 2.3.2.1. Consider the language $X = \{0, 11, 0100\}$. The language $L(X)$ is the language consisting of all strings of length 1, 2, and 4.

Our eventual goal will be to construct a language X such that $L(X) \in \mathsf{NP}^X \setminus \mathsf{P}^X$. Of particular note is that we can rather nicely put a upper bound on the complexity of $L(X)$ when given X as an oracle, regardless of the value of X. This fact is what gives us the freedom to construct X in such a way that $L(X)$ will not be in P^X.

Lemma 2.3.3 ([5, p. 436])**.** *For any oracle X, $L(X) \in \mathsf{NP}^X$.*

Proof. Let S be a string of length n. If $S \in L(X)$, then a witness for S is any string S' such that $|S| = |S'|$ and $S' \in X$. Since a machine with query access to X can query whether S' is in X in one step, it follows that we can verify that $S \in L(X)$ in polynomial time. □

With this lemma as a base, we can now move on to our main theorem.

Theorem 2.3.4 ([5, Theorem 3])**.** *There exists an oracle A such that $\mathsf{P}^A \neq \mathsf{NP}^A$.*

Proof. Our goal is to construct a set B such that $L(B) \notin \mathsf{P}^B$. We shall construct B in an interative manner. We do this by taking a sequence $\{P_i\}$ of all machines that recongize some language in P^A, and then constructing B such that for each machine in the sequence, there is some part of $L(B)$ it cannot recognize. This technique is called *diagonalization*, and it is used in many places in computer science theory.[1] Additionally, we define $p_i(n)$ to be the maximum running time of P_i on an input of length n. We give the following algorithm to construct B:

```
   Input: A sequence of P oracle machines $\{P_i\}_{i=1}^{\infty}$
   Output: A set $B$ such that $L(B) \notin \mathsf{P}^B$
 1 $B(0) \leftarrow \varnothing$;
 2 $n_0 \leftarrow 0$;
 3 for $i$ starting at 1 do
 4     Let $n > n_i$ be large enough that $p_i(n) < 2^n$;
 5     Run $P_i^{B(i-1)}$ on input $0^n$;
 6     if $P_i^{B(i-1)}$ rejects $0^n$ then
 7         Let $x$ be a string of length $n$ not queried during the above computation;
 8         $B(i) \leftarrow B(i-1) \sqcup \{x\}$;
 9     end
10     $n_{i+1} \leftarrow 2^n$;
11 end
12 $B \leftarrow \bigcup_i B(i)$;
```

Algorithm 1: An algorithm for constructing B

Now that we have presented the algorithm, let us demonstrate its soundness. First, note that since P_i runs in polynomial time, $p_i(n)$ is bounded above by a polynomial, and hence there will always exist an n as defined in line 4. Next, since there are 2^n strings of length n and since $p_i(n) < 2^n$, we know that there must be some x to make line 7 well-defined. While our algorithm allows x to be any

[1]This argument style is named after *Cantor's diagonal argument*, which was originally used to prove that the real numbers are uncountable [18, Thm. 2.14].

string, if it is necessary to be explicit in which we choose, then picking x to be the smallest string in lexicographic order is a standard choice.

We should also briefly mention that this algorithm does not terminate. This is okay because we are only using it to construct the set B, which does not need to be bounded. If this were to be made practical, since the sequence of n_is is monotonically increasing, the set could be constructed "lazily" on each query by only running the algorithm until n_i is greater than the length of the query.

Next, we demonstrate that $L(B) \notin \mathsf{P}^B$. The end goal of our instruction is a set B such that if P_i^B accepts 0^n then there are no strings of length n in B, and if P_i^B rejects, then there is a string of length n in B. This means that no P_i accepts $L(B)$, and hence $L(B) \notin \mathsf{NP}^B$.

The central idea behind the proper functioning of our algorithm is that adding strings to our oracle *cannot change the output if they are not queried.* This is what we do in line 4: we need our input length to be long enough to guarantee that a non-queried string exists. Since the number of queried strings is no greater than $p_i(n)$, and there are 2^n strings of length n, there must be some string not queried.

Next, we run $P_i^{B(i-1)}$ on all the strings we have already added. If it accepts, then we want to make sure that no string of length n is in B; that is, 0^n is not in $L(B)$. Hence, in this particular loop we add nothing to $B(i)$. If $P_i^{B(i-1)}$ rejects, we then need to make sure that $0^n \in L(B)$ but in a way that does not affect the output of $P_i^{B(i-1)}$. Hence, we find a string that $P_i^{B(i-1)}$ did not query (and thus will not affect the result) and add it to $B(i)$.

Having done this, we then set n_{i+1} to be 2^n. Since $p_i(n) < 2^n$, it follows that no previous machine could have queried any strings of length n_{i+1}.[2] This way, we ensure our previous machines do not accidentally have their output change due to us adding a string they queried.

Having run this over all polynomial-time Turing machines, we have a set $L(B)$ such that no machine in P^B accepts it, which tells us $L(B) \notin \mathsf{P}^B$. But, Lemma 2.3.3 already told us $L(B) \in \mathsf{NP}^B$. Hence, $\mathsf{P}^B \neq \mathsf{NP}^B$. □

2.4 Diagonalization relativizes

Of course, determining that P vs NP does not relativize is only important if the proof techniques used in practice *do* in fact relativize. Rather unfortunately, it turns out that simple diagonalization is a relativizing result.

While diagonalization itself does not have a formal definition, we can still think about it informally. Looking at our construction of B, which we did using diagonalization, notice that our definition never really cared about how the P_i worked, just about the results it produced. Hence, if it were to be possible to modify Algorithm 1 to construct $B \in \mathsf{NP} \setminus \mathsf{P}$, the proof would remain the same if we were to replace our sequence $\{P_i\}$ with a sequence of machines in P^A for some PSPACE-complete A. However, this would lead to a contradiction, as we showed in Theorem 2.3.1 that in that case, $\mathsf{P}^A = \mathsf{NP}^A$! This tells us that a simple diagonalization argument would not suffice to determine separation between P and NP.

[2] A word of caution: we only care about what P_i does on input n_i, *not any other input.* This is because we only need each machine to be incorrect for some i, not all i.

Bibliography

[1] Scott Aaronson. “$P \overset{?}{=} NP$”. In: *Open Problems in Mathematics*. Ed. by John Forbes Nash Jr. and Michael Th. Rassias. Cham: Springer International Publishing, 2016, pp. 1–122. ISBN: 978-3-319-32162-2. DOI: 10.1007/978-3-319-32162-2_1. URL: https://link.springer.com/chapter/10.1007/978-3-319-32162-2_1.

[2] Scott Aaronson and Avi Wigderson. “Algebrization: A New Barrier in Complexity Theory”. In: *ACM Trans. Comput. Theory* 1.1 (Feb. 2009). ISSN: 1942-3454. DOI: 10.1145/1490270.1490272.

[3] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. 1st. USA: Cambridge University Press, 2009. ISBN: 978-0-521-42426-4. DOI: 10.5555/1540612.

[4] L. Babai, L. Fortnow, and C. Lund. “Nondeterministic exponential time has two-prover interactive protocols”. In: *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*. 1990, 16–25 vol.1. DOI: 10.1109/FSCS.1990.89520.

[5] Theodore Baker, John Gill, and Robert Solovay. “Relativizations of the $\mathcal{P} \overset{?}{=} \mathcal{NP}$ Question”. In: *SIAM Journal on Computing* 4.4 (1975), pp. 431–442. DOI: 10.1137/0204037. eprint: https://doi.org/10.1137/0204037. URL: https://doi.org/10.1137/0204037.

[6] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. “Alternation”. In: *J. ACM* 28.1 (Jan. 1981), pp. 114–133. ISSN: 0004-5411. DOI: 10.1145/322234.322243. URL: https://dl.acm.org/doi/10.1145/322234.322243.

[7] Alessandro Chiesa et al. “Spatial Isolation Implies Zero Knowledge Even in a Quantum World”. In: *J. ACM* 69.2 (Jan. 2022). ISSN: 0004-5411. DOI: 10.1145/3511100. URL: https://doi.org/10.1145/3511100.

[8] Stephen A. Cook. “The complexity of theorem-proving procedures”. In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC ’71. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158. ISBN: 9781450374644. DOI: 10.1145/800157.805047. URL: https://doi.org/10.1145/800157.805047.

[9] Thomas H. Cormen et al. *Introduction to Algorithms*. 4th ed. MIT Press, 2022. ISBN: 978-0-262-04630-5. URL: https://mitpress.mit.edu/9780262046305/introduction-to-algorithms.

[10] Oded Goldreich. *Foundations of Cryptography*. Vol. 1. 1st ed. Cambridge University Press, 2001. 372 pp. ISBN: 978-0-511-54689-1. DOI: 10.1017/CBO9780511546891.

[11] Tom Gur, Jack O’Connor, and Nicholas Spooner. “Perfect Zero-Knowledge PCPs for #P”. In: *Proceedings of the 56th Annual ACM Symposium on Theory of Computing*. STOC 2024. Vancouver, BC, Canada: Association for Computing Machinery, 2024, pp. 1724–1730. ISBN: 9798400703836. DOI: 10.1145/3618260.3649698. URL: https://doi.org/10.1145/3618260.3649698.

[12] Kenneth E. Iverson. *A Programming Language*. 1st ed. John Wiley and Sons, Inc., 1962. 286 pp. ISBN: 978-0471430148. URL: https://dl.acm.org/doi/10.5555/1098666.

[13] Zhengfeng Ji et al. “MIP* = RE”. In: *Commun. ACM* 64.11 (Oct. 2021), pp. 131–138. ISSN: 0001-0782. DOI: 10.1145/3485628. URL: https://dl.acm.org/doi/10.1145/3485628.

[14] Ali Juma et al. “The Black-Box Query Complexity of Polynomial Summation”. In: *Comput. Complex.* 18.1 (Apr. 2009), pp. 59–79. ISSN: 1016-3328. DOI: 10.1007/s00037-009-0263-7. URL: https://doi.org/10.1007/s00037-009-0263-7.

[15] Donald E. Knuth. “Two Notes on Notation”. In: *The American Mathematical Monthly* 99.5 (1992), pp. 403–422. ISSN: 00029890, 19300972. URL: http://www.jstor.org/stable/2325085 (visited on 11/19/2024).

[16] Carsten Lund et al. “Algebraic methods for interactive proof systems”. In: *J. ACM* 39.4 (Oct. 1992), pp. 859–868. ISSN: 0004-5411. DOI: 10.1145/146585.146605. URL: https://doi.org/10.1145/146585.146605.

[17] Kieran Mastel and William Slofstra. “Two Prover Perfect Zero Knowledge for MIP*”. In: *Proceedings of the 56th Annual ACM Symposium on Theory of Computing.* STOC 2024. Vancouver, BC, Canada: Association for Computing Machinery, 2024, pp. 991–1002. ISBN: 9798400703836. DOI: 10.1145/3618260.3649702. URL: https://doi.org/10.1145/3618260.3649702.

[18] Walter Rudin. *Principles of Mathematical Analysis.* 3rd ed. McGraw-Hill, 1976. 342 pp. ISBN: 978-0-07-085613-4.

[19] Walter J. Savitch. “Relationships between nondeterministic and deterministic tape complexities”. In: *Journal of Computer and System Sciences* 4.2 (1970), pp. 177–192. ISSN: 0022-0000. DOI: 10.1016/S0022-0000(70)80006-X. URL: https://www.sciencedirect.com/science/article/pii/S002200007080006X.

[20] Michael Sipser. *Introduction to the Theory of Computation.* 1st ed. International Thomson Publishing, 1996. 396 pp. ISBN: 978-0-534-94728-6. DOI: 10.1145/230514.571645.

[21] L. J. Stockmeyer and A. R. Meyer. “Word problems requiring exponential time”. In: *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing.* STOC ’73. Austin, Texas, USA: Association for Computing Machinery, 1973, pp. 1–9. ISBN: 9781450374309. DOI: 10.1145/800125.804029. URL: https://doi.org/10.1145/800125.804029.

[22] A. M. Turing. “On Computable Numbers, with an Application to the Entscheidungsproblem”. In: *Proceedings of The London Mathematical Society* 42.1 (1936), pp. 230–265. DOI: 10.1112/PLMS/S2-42.1.230.