Complexity Limitations on Quantum Computation

Lance Fortnow*
University of Chicago
Department of Computer Science
1100 E. 58th St.
Chicago, IL 60637

John Rogers†
School of CTI
DePaul University
243 S. Wabash Ave.
Chicago, IL 60604

December 4, 1997

Abstract

We use the powerful tools of counting complexity and generic oracles to help understand the limitations of the complexity of quantum computation. We show several results for the probabilistic quantum class **BQP**.

- **BQP** is low for **PP**, i.e., $\mathbf{PP}^{\mathbf{BQP}} = \mathbf{PP}$.
- There exists a relativized world where $\mathbf{P} = \mathbf{BQP}$ and the polynomial-time hierarchy is infinite.
- There exists a relativized world where $\mathbf{P} = \mathbf{BQP}$ but $\mathbf{P} \neq \mathbf{UP} \cap \mathbf{coUP}$ and one-way functions exist. This gives a relativized answer to an open question of Simon.

1 Introduction

We have seen a surge in interest in quantum computation over the past few years. This interest comes from new and good theoretical models for quantum Turing machines [BV97] and surprising algorithms for factoring [Sho97] and searching [Gro96].

Exactly how much can we accomplish with quantum computers? We bring two powerful tools from computational complexity theory to bear on this question. First we use counting complexity, in particular the **GapP** functions developed by Fenner, Fortnow and Kurtz [FFK94], to give us new upper bounds on quantum complexity classes. Next we use generic oracles to show that the existence of one-way functions does not necessarily imply that quantum computers are more powerful than deterministic ones.

The notion of efficient computation is captured by the bounded probabilistic polynomial-time Turing machine. Such a machine accepts an input x either with probability greater than or equal to $2/3$ or with probability less than or equal to $1/3$. In the first case, we say that x is in the language accepted by M and in the second that it is not. The class of languages accepted by these machines is called **BPP**. If we replace the Turing machine with a *quantum* Turing machine, the class of accepted languages is called **BQP**.

The power of a quantum Turing machine lies in its ability to have its superpositions "cancel" each other. Fenner, Fortnow and Kurtz [FFK94] developed the notion of **GapP** functions, the closure of the **#P** functions under subtraction. The **GapP** functions have some powerful applications based on a similar cancellation effect. We show, perhaps not too surprisingly, a close relationship between **GapP**-definable counting classes and quantum computing. We can use this relationship to obtain new limitations on the complexity of quantum computing.

*Email: fortnow@cs.uchicago.edu. URL: http://www.cs.uchicago.edu/~fortnow.

Supported in part by NSF grant CCR 92-53582. Some of this research occurred while the author was visiting the CWI in Amsterdam.

†Email: rogers@cs.depaul.edu. URL: http://www.depaul.edu/~jrogers.

We show that **BQP** sits in the counting class **AWPP**. Based on a previous result about **AWPP** [Li93], we can show that **BQP** is low for **PP** and so improve the upper bound given by Bennet, Bernstein, Brassard and Vazirani [BBBV97]. We can also use oracle results about **AWPP** to get a relativized world where **P** = **BQP** but the polynomial-time hierarchy is infinite.

We know that **BPP** $\subseteq$ **BQP**. An important open question is whether or not the containment is strict. Answering this question will probably be very difficult in that, if the containment is strict, we can show a separation between **BPP** and **PSPACE**. One approach is to investigate what kinds of conditions would cause a separation between **BPP** and **BQP**. For example, Simon [Sim97] asked whether the existence of one-way functions is sufficient to cause a separation. A one-way function is a one-to-one, honest, polynomial-time computable function whose inverse is not polynomial-time computable.

Our result shows that there is a relativized world in which this implication fails. Although this does not directly refute the implication, it does demonstrate that, if the implication is true, proving it will require techniques that do not relativize.

2 Definitions

2.1 Preliminaries

As usual, Σ denotes the alphabet $\{0,1\}$ and Σ^* the set of all finite length strings over Σ. A *language* is a subset of Σ^*.

2.2 Models of computation

Our models of computation are the (classical) Turing machine and the quantum Turing machine. Unless otherwise stated, we assume that all machines are polynomial-time bounded. See Hopcroft and Ullman [HU79] for definitions regarding classical Turing machines.

We allow the machines to have oracle access, which means that they are allowed to make membership queries to a language A called the oracle and to receive a correct response in constant time. Such machines have a separate query tape and three extra states: a *query* state, a *yes* state, and a *no* state. When a machine is computing relative to an oracle A, it can query whether a string x is in A by writing x on the query tape and entering the query state. If $x \in A$, the computation's next state is the *yes* state, otherwise it is the *no* state.

A language L is in the complexity class **BPP** if there is a classical machine M such that, for every $x \in L$, at least $2/3$ of the paths in the computation $M(x)$ are accepting paths and, for every $x \notin L$, no more than $1/3$ of the paths are accepting paths.

2.3 Quantum Computation

We will use a simplified model of quantum computation due to Bernstein and Vazirani [BV97]. While simple, this model captures all of the power of quantum computation. See the paper by Bernstein and Vazirani [BV97] for a discussion of this model and the physics underlying it. For a general introduction to quantum computation see the survey by Berthiaume [Ber97].

Consider the transition function of a Turing machine that maps current state and contents under the tape heads to a new state, new values to write under the tape heads and a direction to move the heads. A deterministic Turing machine's transition function has a single output. A probabilistic Turing machine's transition function maps to a distribution on outputs with nonnegative probabilities that add up to one.

A quantum Turing machine's transition function maps to a *superposition* of the outputs where each output gets an *amplitude* which may be a complex value. To simplify matters, Adleman, DeMarrais and Huang [ADH97] and Solovay and Yao [SY96] show that we can assume these amplitudes take one of the values in the set $\{-1, -\frac{4}{5}, -\frac{3}{5}, 0, \frac{3}{5}, \frac{4}{5}, 1\}$.

The quantum Turing machines we consider here all run in polynomial time and thus have an exponential number of possible configurations.

Suppose that before a transition each configuration C_i has a real amplitude α_i. Consider the L_2 norm of the amplitudes

$$\sqrt{\sum_i \alpha_i^2}$$

A quantum Turing machine is required to preserve this L_2 norm. This is equivalent to the transition matrix U of the configurations being unitary. For real U, U is unitary if the transpose of U is the inverse of U.

To compute the probability of acceptance consider an initial configuration amplitude vector $\vec{\alpha}$ where $\alpha_0 = 1$ for the initial configuration C_0 and $\alpha_i = 0$ for each other configuration. Let $\vec{\beta} = U^t \cdot \vec{\alpha}$ where t is the running time of the Turing machine. The probability of acceptance is the sum of the squares of the β_i where C_i is an accepting configuration.

We can now define **BQP** similar to the definition of **BPP**.

Definition 2.1 *A language L is in* **BPP** *if there is a quantum Turing machine M such that for all x in Σ^*,*

- *If x is in L then $M(x)$ accepts with probability at least two-thirds.*
- *If x is not in L then $M(x)$ accepts with probability at most one-third.*

Similar to **BPP** though with nontrivial proofs, we can assume the error is exponentially small and that **BQP** machines can simulate deterministic Turing machines and other **BQP** machines (see [BV97]).

We can also define the class **EQP** or **QP** which has the same definition as **BQP** except that we require zero error. Surprisingly, this class appears to be stronger than deterministic polynomial time (see [BV97]).

2.4 Counting Classes

A function f is in the counting class **#P** if there is a nondeterministic polynomial-time machine M such that $f(x) = m$ iff the computation $M(x)$ has m accepting paths. In order to understand better the complexity of counting classes like **#P**, Fenner, Fortnow and Kurtz [FFK94] defined the **GapP** functions consisting of the closure under subtraction of the set of **#P** functions. Equivalently, **GapP** consists of functions $f(x)$ such that for some nondeterministic polynomial-time Turing machine M, $f(x)$ is the difference of the number of accepting and the rejecting paths of $M(x)$.

The power of **GapP** functions lie in their closure properties: **GapP** functions are closed under negation, subtraction, exponential addition and polynomial multiplication.

Theorem 2.2 (Fenner-Fortnow-Kurtz) *Let f be a* **GapP** *function and q a polynomial. Then the following are* **GapP** *functions:*

1. *$-f(x)$,*
2. *$\sum_{|y| \leq q(|x|)} f(x,y)$, and*
3. *$\prod_{0 \leq y \leq q(|x|)} f(x,y)$.*

We can also define many interesting counting classes using **GapP** functions. For this paper we consider the following classes.

Definition 2.3 *The class* **PP** *consists of those languages L such that for some* **GapP** *function f and all x in Σ^*,*

- *If x is in L then $f(x) > 0$.*
- *If x is not in L then $f(x) < 0$.*

The class **PP** was first defined by Gill [Gil77] as probabilistic polynomial time with unbounded error. Definition 2.3 first given by Fenner, Fortnow and Kurtz [FFK94] makes the class considerably easier to work with.

Definition 2.4 *The class* **LWPP** *consists of those languages L such that for some* **GapP** *function f a polynomial-time computable positive function g and all x in Σ^*,*

- *If x is in L then $f(x) = g(1^{|x|})$.*
- *If x is not in L then $f(x) = 0$.*

Definition 2.5 *The class* **AWPP** *consists of those languages L such that for all polynomials q, there is a* **GapP** *function f and a polynomial-time computable function g such that for all strings x in Σ^* and $m \geq |x|$, $0 < f(x, 1^m) < g(1^m)$ and*

- *If x is in L then $f(x, 1^m) \geq (1 - 2^{-q(m)})g(1^m)$.*
- *If x is not in L then $f(x, 1^m) \leq 2^{-q(m)}g(1^m)$.*

The classes **LWPP** and **AWPP** were first defined by Fenner, Fortnow and Kurtz [FFK94] and Fenner, Fortnow, Kurtz and Li [FFKL93]. Though artificial, these classes have some nice properties that we will use to help classify quantum complexity.

2.5 One-way functions

A language L is in the class **UP** if there is a classical machine M that, for every $x \in L$, has exactly one accepting path.

A polynomial-time computable function $f : \Sigma^* \to \Sigma^*$ is *one-way* if it is one-to-one, honest, and not invertible. Being *honest* means that there is a polynomial p such that $p(|f(x)|) > |x|$; in other words, honest functions do not map a long input string to a short output string.

Grollmann and Selman [GS88] showed that one-way functions exist if and only if $\mathbf{P} \neq \mathbf{UP}$.

2.6 Generic oracles

In trying to show that there is an oracle relative to which a particular proposition P holds, we often begin by defining an infinite set of *requirements*, which are statements about relativized computations. An oracle X *satisfies* (or *forces*) a requirement if the statement of the requirement is true when the computations are performed relative to X. We define the requirements so that, if each is satisfied, the proposition P is true. For example, to make $\mathbf{P}^X \neq \mathbf{NP}^X$, we specify an enumeration of all polynomial-time bounded, deterministic oracle Turing machines: $\{M_i\}_{i \in \omega}$. We then define a nondeterministic machine N and an infinite set of requirements $R = \{R_i\}_{i \in \omega}$, where R_i is the statement: "$L(M_i^X) \neq L(N^X)$." If we construct an oracle X satisfying every R_i then $\mathbf{P}^X \neq \mathbf{NP}^X$.

Defining the requirements is often quite straightforward. The difficulties usually arise when trying to construct the oracle. We avoid some of the difficulties by only employing *generic* oracles.

A *condition* is a partial function from Σ^* to $\{0, 1\}$. A condition σ *extends* another condition τ if, for all $x \in \mathrm{d}om(\tau)$, $\sigma(x) = \tau(x)$. Two conditions σ and τ are *compatible* if, for all $x \in \mathrm{d}om(\sigma) \cap \mathrm{d}om(\tau)$, $\sigma(x) = \tau(x)$. They *conflict* otherwise. We will always assume that if a condition is defined on any string of some length then it is defined on all strings of that length.

A condition σ *satisfies* a requirement if the oracle $\overline{\sigma}$ satisfies it.

A set of conditions S is *dense* if, for every condition τ, there is a condition $\sigma \in S$ that extends τ. It is *definable* if the set $\{\overline{\sigma} : \sigma \in S\}$ belongs to Π_1^1 (see [Rog87]).

Restrictions can be set on conditions to achieve a desired separation. The restrictions we impose in this paper are that a condition has a finite domain and that a condition can take the value 1 on at exactly one string of every *acceptable length* and it takes on the value 0 for any string not of an acceptable length. An acceptable length is an integer in the range of the *tower* function, which has the recursive definition: $tower(0) = 2$, $tower(n+1) = 2^{tower(n)}$. That is, $tower(n)$ is a tower of 2's with height $n+1$. Conditions with these restrictions are called **UP** $\cap$ **coUP**-*conditions*.

An oracle A *meets* a set of conditions S if there is some σ in S that is extended by the characteristic function for A. A *generic* oracle is one that meets every dense definable set of conditions (unrestricted). A **UP** $\cap$ **coUP**-*generic* oracle is one that meets every dense definable set of **UP** $\cap$ **coUP**-conditions.

UP$\cap$**coUP**-generics were first developed by Fortnow and Rogers [FR94] to study the relationship between separability and one-way functions.

3 Quantum Computing and Counting Complexity

In this section we show a close connection between counting complexity and quantum computing.

Theorem 3.1

$$\mathbf{BQP} \subseteq \mathbf{AWPP}$$

Proof. Fix a language L in **BQP** and a polynomial q. Let M be a polynomial-time quantum Turing machine that on input $(x, 1^m)$ accepts for x in L with probability at least $1 - 2^{-q(m)}$ and accepts for x not in L with probability at most $2^{-q(m)}$.

Fix x and m with $m \geq |x|$.

Let t be the running time of M. We can assume that t is a polynomial in m and that M has at most 2^t configurations. Let U be the transition matrix of M. By the discussion in Section 2.3 we can assume the entries of U are of the form $w/5$ for w an integer between -5 and 5. By the nature of a transition matrix, we can compute the (i, j) entry of U in time polynomial in m.

Let $V = 5U$ so V has only integral entries. Let $\vec{\alpha}$ be the initial configuration amplitude vector as described in Section 2.3. Let $\vec{\beta} = V^t \cdot \vec{\alpha}$. Note that each β_i, a component of $\vec{\beta}$ corresponding to configuration C_i, is an exponential sum of a polynomial product of polynomial-time computable entries of V. By Theorem 2.2, we have that each β_i is a **GapP** function.

Let $f(x, m)$ be the sum of the squares of β_i where C_i is an accepting configuration. Again by Theorem 2.2 we have $f(x, m)$ is a **GapP** function. Also we have that $\frac{f(x,m)}{(5^t)^2}$ is the acceptance probability of $M(x, m)$. We can thus fulfill the requirements of Definition 2.5 by letting $g(1^m) = 5^{2t}$. ∎

We can now use properties of **AWPP** to better understand the complexity of **BQP**. Lide Li [Li93] gave an upper bound on the complexity of **AWPP**.

Theorem 3.2 (Li) AWPP *is low for* **PP**, *i.e.,* $\mathbf{PP}^{\mathbf{AWPP}} = \mathbf{PP}$.

For completeness we sketch the proof of Theorem 3.2.

Proof Sketch. Suppose L is in $\mathbf{PP}^A$ for some A in **AWPP**. By Definition 2.3, there is some $h \in \mathbf{GapP}^A$ such that for $x \in L$, $h(x) \geq 1$ and $h(x) \leq -1$ otherwise. Let M^A be a relativized nondeterministic polynomial-time Turing machine such that $h(x)$ is the difference of the number of accepting and rejecting computations of M^A.

Pick a polynomial $q(n)$ such that for strings of length n, M^A has less than $2^{q(n)/2}$ computation paths. Let f and g be **GapP** and polynomial-time computable functions defined for A and q as in Definition 2.5. Let N be a nondeterministic polynomial-time Turing machine such that $f(x, 1^m)$ is the difference of the number of accepting and rejecting paths of $N(x, 1^m)$.

Create a new nondeterministic polynomial-time Turing machine M' as follows. On input x, simulate $M^A(x)$. Every time M makes a query y to A, simulate $N(y, 1^{|x|})$. If N accepts then continue the computation of M assuming y is in A. If N rejects then continue the computation of M assuming y is not in A.

By the choice of q, the mistakes made by the wrong simulation, even totaled over every computation path of $M^A(x)$, is not enough to effect the sign of the difference of the number of accepting and rejecting paths of M'. ∎

From Theorem 3.2 we get the same result for **BQP**.

Corollary 3.3 **BQP** *is low for* **PP**.

This improves and simplifies the bound given by Adleman, DeMarrais and Huang [ADH97].

Corollary 3.4 (Adleman-DeMarrais-Huang)

$$\mathbf{BQP} \subseteq \mathbf{PP} \subseteq \mathbf{P}^{\#\mathbf{P}} \subseteq \mathbf{PSPACE}$$

We also have a class containing **BQP** that is not known to contain **NP** as Beigel [Bei94] has a relativized world where **NP** is not low for **PP**.

Fenner, Fortnow, Kurtz and Li [FFKL93] give an interesting collapse for **AWPP** relative to generic oracles. Their proof builds on a connection between decision tree complexity and low-degree polynomials developed by Nisan and Szegedy [NS92].

Theorem 3.5 *If* $\mathbf{P} = \mathbf{PSPACE}$ *(unrelativized) then* $\mathbf{P}^G = \mathbf{AWPP}^G$ *for any generic* G*.*

We can create an oracle H by starting with an oracle making $\mathbf{P} = \mathbf{PSPACE}$ and joining a generic G to that. Combine this with the fact that Theorem 3.1 relativizes gives us some interesting relativized worlds.

Corollary 3.6 *There exists a relativized world where* $\mathbf{P} = \mathbf{BQP}$ *and the polynomial-time hierarchy is infinite.*

This greatly strengthens the result of Bennet, Bernstein, Brassard and Vazirani [BBBV97] giving a relativized world where **NP** is not in **BQP**.

Using a proof similar to that of Theorem 3.1 we get an stronger upper bound for **EQP**.

Theorem 3.7

$$\mathbf{EQP} \subseteq \mathbf{LWPP}$$

Whether Graph Isomorphism can be solved quickly by quantum computers remains an interesting open question. This possibility is consistent with Theorem 3.7 as Köbler, Schöning and Torán [KST92] have show that Graph Isomorphism sits in **LWPP**.

4 Quantum Computing and One-Way Functions

We show that one-way functions are not sufficient to guarantee the hardness of **BQP**.

Theorem 4.1 *There is an oracle* C *relative to which one-way functions exist and* $\mathbf{P}^C = \mathbf{BQP}^C$*.*

Thus, to demonstrate that the existence of one-way functions implies a separation between **BPP** and **BQP** will require nonrelativizing techniques.

We actually prove a stronger result from which Theorem 4.1 follows.

Theorem 4.2 *There is an oracle* C *relative to which* $\mathbf{P} = \mathbf{BPP} = \mathbf{BQP} \neq \mathbf{UP} \cap \mathbf{coUP}$*.*

To prove Theorem 4.2, we need the following theorem due to Bennett, Bernstein, Brassard and Vazirani [BBBV97].

Theorem 4.3 *Given an oracle* **BQP** *machine* M *that runs in time* $p(n)$*, an oracle* O*, and an* n*-bit string* x*, it is the case that, for all but* $2p^2(n)/\epsilon$ *strings* y*, the oracle* O'*, where* $O'(z) = O(z)$ *iff* $z \neq y$*, is such that* $|P[M^{O'} \text{ accepts } x] - P[M^O \text{ accepts } x]| \leq \epsilon$*.*

This theorem states that for an oracle **BQP** Turing machine M and an input x whose length is n, there is a polynomial (in n) sized set S such that, if a string y is not in S, we can change the oracle's answer on y and the probability that M accepts x is still bounded away from 1/2.

Proof of Theorem 4.2. Let H be an oracle relative to which $\mathbf{P} = \mathbf{PSPACE}$ (H can be any **PSPACE**-complete language). Let G be a $\mathbf{UP} \cap \mathbf{coUP}$-generic oracle, which allows at most one string at lengths that are exponentially far apart. Let $C = H \oplus G = \{0x | x \in H\} \cup \{1y | y \in G\}$. In effect, C represents a relativization that identifies **P** and **PSPACE** (and so $\mathbf{P} = \mathbf{BPP} = \mathbf{BQP}$) and a *re-relativization* that, we will show, separates **P** and $\mathbf{UP} \cap \mathbf{coUP}$ but that still leaves $\mathbf{P} = \mathbf{BPP} = \mathbf{BQP}$.

First we show that $\mathbf{P}^C \neq \mathbf{UP}^C \cap \mathbf{coUP}^C$. Let $L^X = \{0^n | (\exists x) |x| = n \ \&\ x \in X\}$. It's easy to see that $L^G \in \mathbf{UP}^G \cap \mathbf{coUP}^G$ and so in $\mathbf{UP}^C \cap \mathbf{coUP}^C$. A diagonalization argument will demonstrate that $L^G \notin \mathbf{P}^G$. Because G is generic with respect to H, $L^C \notin \mathbf{P}^C$.

Next we show that $\mathbf{P}^C = \mathbf{BQP}^C$. Let M be a categorically **BQP** machine that runs in time $p(n)$. Being categorically **BQP** means that no matter what strings are in the oracle G, M will always accept with probability greater than or equal to 2/3 or less than or equal to 1/3. Let x be an input of length n.

We need to show that there is a deterministic polynomial-time machine N that, relative to C, determines for an input x whether $M^C(x)$ accepts. Because M runs in polynomial time, there are at most a polynomial number of lengths for strings that M can query in an oracle. Because G has at most one string at every length, there are at most polynomially many strings in it that could affect M's computation on x. Because the strings in G are exponentially far apart, all but at most one are at lengths that are so short that N on input x can query G on every string at those lengths and so find all of them.

So the only string that N needs to worry about is one at a length ℓ that is so large that N would have to query exponentially many strings to be certain of finding it. Call this string y.

But N *can* figure out what M would do on input x under the assumption that there are no strings of length ℓ in G. Let us say that, under this assumption, $M(x)$ accepts. If in fact there are no strings of length ℓ in G then N would be right in accepting x. However, if there is a string of length ℓ in G, this string could cause M to change its mind and reject x. But theorem 4.3 says that there is a set of strings S whose cardinality is bounded by $2p^2(n)/\epsilon$ such that, if y is not in S, the probability that M changes its mind is less than or equal to ϵ.

Set ϵ to a value strictly less than 1/6 and say that we know that y (if it exists) is not in S. If there are no strings in G of length ℓ, we know that M accepts x with probability at least 2/3. If there is a string y of length ℓ in G and if that string is not in S, theorem 4.3 tells us that the probability that M accepts x is still strictly greater than 1/2. In other words, x is still in the language accepted M relative to G. So if N knows that y, if it exists, is not in S, it can simply run the simulation of $M(x)$ under the assumption that there is no string of length ℓ in G and output the correct answer.

So how does N determine whether y is in S? It asks for an explicit enumeration of S. That is, it asks the question: "What is the set S of strings of length ℓ such that, if one of those strings is in the oracle, M rejects x?" S has size $2cp^2(n)$, where $c > 6$.

This question can be answered in **PSPACE** without querying G. N can use its access to H to find S in polynomial time. It then queries G for each of those strings. If none of those strings are in G then N accepts input x because that is what M would do. If N finds one of those strings in G, it would then be able to simulate the computation of $M(x)$ with full knowledge of all of the strings in G that could possibly affect that computation. ∎

5 Conclusions

We give results in this paper indicating severe restrictions on the complexity of quantum computing. We conjecture that **BQP** actually contains *no* interesting complexity classes outside of **BPP**.

Still we believe that quantum computing remains a potentially powerful model of computation. Quantum computers can quickly solve some problems not known complete such as factoring [Sho97] and the potential to solve problems such as graph isomorphism and finding a short vector in a lattice. Also quantum computing can give a large increase in speed, for example a quadratic improvement in **NP**-like search problems [Gro96].

Acknowledgments

The authors would like to thank André Berthiaume, Harry Buhrman, Richard Cleve and Wim van Dam for a number of illuminating conversations on quantum computation.

References

[ADH97] L. Adleman, J. DeMarrais, and M. Huang. Quantum computability. *SIAM Journal on Computing*, 26(5):1524–1540, 1997.

[BBBV97] C. Bennett, E. Bernstein, G. Brassard, and U. Vazirani. Strengths and weaknesses of quantum computing. *SIAM Journal on Computing*, 26(5):1510–1523, 1997.

[Bei94] R. Beigel. Perceptrons, PP and the polynomial hierarchy. *Computational Complexity*, 4:314–324, 1994.

[Ber97] A. Berthiaume. Quantum computation. In A. Selman and L. Hemaspaandra, editors, *Complexity Theory Retrospective II*, pages 23–51. Springer, 1997.

[BV97] E. Bernstein and U. Vazirani. Quantum complexity theory. *SIAM Journal on Computing*, 26(5):1411–1473, 1997.

[FFK94] S. Fenner, L. Fortnow, and S. Kurtz. Gap-definable counting classes. *Journal of Computer and System Sciences*, 48(1):116–148, 1994.

[FFKL93] S. Fenner, L. Fortnow, S. Kurtz, and L. Li. An oracle builder's toolkit. In *Proceedings of the 8th IEEE Structure in Complexity Theory Conference*, pages 120–131. IEEE, New York, 1993.

[FR94] L. Fortnow and J. Rogers. Separability and one-way functions. In *Proceedings of the 5th Annual International Symposium on Algorithms and Computation*, volume 834 of *Lecture Notes in Computer Science*, pages 396–404. Springer, Berlin, 1994.

[Gil77] J. Gill. Computational complexity of probabilistic complexity classes. *SIAM Journal on Computing*, 6:675–695, 1977.

[Gro96] L. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the 28th ACM Symposium on the Theory of Computing*, pages 212–219. ACM, New York, 1996.

[GS88] J. Grollmann and A Selman. Complexity measures for public-key cryptosystems. *SIAM Journal on Computing*, 17:309–355, 1988.

[HU79] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, Mass., 1979.

[KST92] J. Köbler, U. Schöning, and J. Torán. Graph isomorphism is low for PP. *Computational Complexity*, 2(4):301–330, 1992.

[Li93] L. Li. *On the counting functions*. PhD thesis, University of Chicago, 1993. Department of Computer Science TR 93-12.

[NS92] N. Nisan and M. Szegedy. On the degree of boolean functions as real polynomial. In *Proceedings of the 24th ACM Symposium on the Theory of Computing*, pages 462–467. ACM, New York, 1992.

[Rog87] H. Rogers. *Theory of Recursive Functions and Effective Computability.* MIT Press, Cambridge, Massachusetts, 1987.

[Sho97] P. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.

[Sim97] D. Simon. On the power of quantum computation. *SIAM Journal on Computing*, 26(5):1474–1483, 1997.

[SY96] R. Solovay and A. Yao, 1996. Manuscript.