# Week 1

# Query Complexity Basics

Welcome to CS867 / QIC890: Quantum Query and Communication Complexity. In this course, we will study two commonly-studied models of quantum computation. The query model is the one we will start with. This is one of the simplest possible models of computation, but despite its simplicity, it turns out that most quantum algorithms can be fruitfully analyzed in the query complexity setting. This includes Shor's algorithm and Grover's algorithm, both of which can naturally be viewed as quantum query algorithms.

For this first week, we will introduce the query complexity model, define some basic query measures (including quantum query complexity), and go over some relationships and separations between them. We will start by defining (classical) query complexity.

## 1.1 Classical query complexity

In query complexity, also known as the blackbox model, we are given as input a blackbox $x$ computing some unknown function. We would like to compute some property $f(x)$ of this blackbox $x$. What we are allowed to do is to make *queries* to the blackbox: we can feed in an input $i$ to $x$, and observe the output $x(i)$. Our goal is to compute $f(x)$ using as few such queries to $x$ as possible.

Let $n$ be the number of different types of inputs $x$ accepts (that is, the size of the domain of $x$). Then the function $x$ is completely determined by the answer to these $n$ queries. Without loss of generality, suppose that the domain of $x$ is simply the set $\{1, 2, \ldots, n\}$, which we will denote by $[n]$. We can then think of the function $x$ as a string of length $n$, where position $i$ contains the answer to query $i$. In other words, we will write $x_i$ instead of $x(i)$ to denote the answer $x$ gives on query $i$, and view $x$ as simply an $n$-bit string. Typically, the blackbox $x$ will output Boolean outputs, which makes the string $x$ a Boolean string in $\{0,1\}^n$; however, it is also possible to consider blackboxes with non-Boolean outputs, in which case the string $x$ will be in $\Sigma^n$ for some alphabet $\Sigma$.

Our goal then becomes computing some function $f$ of the input $x$ by reading as few bits of $x$ as possible. More specifically, the function $f$ will be a fixed, known function, typically with Boolean outputs: $f\colon \{0,1\}^n \to \{0,1\}$. For example, $f$ might be the OR function, in which case our goal would be to determine whether the input string $x$ is all zeros (equivalently, whether the input blackbox $x$ ever returns a 1). We care about succeeding even against worst-case inputs $x \in \{0,1\}^n$.

We use $\mathrm{D}(f)$ to denote the minimum number of queries that are required to compute $f$ using a deterministic algorithm.= This is the minimum, over all algorithms which correctly compute $f$ on all inputs, of the worst-case number of queries the algorithm makes against any input $x$.

How shall we formally define $\mathrm{D}(f)$? Well, to do this we need to formally define a deterministic query algorithm. Note that the only thing we care about in an algorithm is which queries it makes

and what its output is. For this reason, a deterministic query algorithm will be defined as a *decision tree*, as follows.

**Definition 1.1.** *A (deterministic) decision tree on inputs of size $n$ is either a leaf in $\{0,1\}$ or a tuple $(i, D_0, D_1)$ where $i \in [n]$ and where $D_0$ and $D_1$ are decision trees (on inputs of size $n$) which do not use $i$ in any internal tuple.*

This definition says that deterministic decision trees (which can also simply be called deterministic query algorithms) are rooted binary trees with internal nodes labeled by $[n]$ and leaves labeled by $\{0,1\}$, with the property that an internal node and its ancestor cannot both have the same label. The way this works is as following: on input $x$, the algorithm starts at the top of the tree, and if the node looks like $(i, D_0, D_1)$, the algorithm queries bit $i$ of $x$ to get the answer $x_i \in \{0,1\}$. The algorithm then goes to $D_0$ if $x_i = 0$ and to $D_1$ if $x_i = 1$, and repeats the same process there; for example, if $x_i = 0$ and $D_0 = (j, D_0', D_1')$, the algorithm will query bit $i$ of $x$ and then query bit $j$ of $x$, and continue on to $D'_{x_j}$. When the algorithm reaches a leaf, it outputs the value of that leaf and halts.

It is not hard to see that such decision trees capture the full generality of deterministic query algorithms (we've assumed here that the alphabet is $\{0,1\}$ and when the function $f$ is Boolean valued, but it is easy to generalize to non-Boolean input and output alphabets in the natural way). Given a decision tree $D$ on $n$ bits and a string $x \in \{0,1\}^n$, we use $D(x)$ to denote the leaf $D$ reaches when run on $x$. We say $D$ *computes* $f$ if $D(x) = f(x)$ for all $x$.

Next, define the height of the decision tree $D$ to be the number of internal nodes in the longest root-to-leaf path in $D$. Observe that this exactly corresponds to the worst-case number of queries $D$ makes on any input. Also observe that since we required a node and its ancestor to have different labels, the height of each decision tree on $n$ bits can be at most $n$.

We can now formally define the deterministic query complexity $\mathrm{D}(f)$ of a Boolean function $f\colon \{0,1\}^n \to \{0,1\}$: it is simply the minimum height of a decision tree computing $f$. Observe that every function has some decision tree computing it, since we can always read all $n$ bits of the input, determine the input $x$, and therefore determine $f(x)$. Since decision trees on $n$ bits have height at most $n$, we conclude that $\mathrm{D}(f) \leq n$ for all $f$. We also have $\mathrm{D}(f) = 0$ if and only if $f$ is constant.

The measure $\mathrm{D}(f)$ captures the query cost of computing $f$ using deterministic algorithms. We will want to contrast this with the cost of computing $f$ using quantum algorithms, but first, we should talk about randomized algorithms.

A *randomized* algorithm can make random queries to the input $x$, and must still compute $f(x)$. One useful trick in computer science is to think of randomized algorithms as deterministic algorithms that depend not only on the input, but also on some register $r$, which will be randomly generated when the algorithm starts. This register $r$ will contain the answers to all the coin flips the algorithm may later wish to toss; we can assume without loss of generality that all these coins were tossed at the beginning.

Now, in the query complexity setting, we observe that for each fixed choice of randomness $r$, the algorithm becomes simply a deterministic algorithm, and hence a deterministic decision tree. It follows that we can model a randomized query algorithm as simply a probability distribution over deterministic decision trees.

**Definition 1.2.** *A randomized query algorithm (or randomized decision tree) on $n$ bits is a probability distribution over deterministic decision trees on $n$ bits.*

Next, we would like to get from this definition of a randomized algorithm to a definition of randomized query complexity $\mathrm{R}(f)$, analogous to $\mathrm{D}(f)$. Recall that $\mathrm{D}(f)$ was the minimum height

of a decision tree computing $f$. We now need to define what it means for a randomized algorithm to compute $f$, and also what the height of a randomized algorithm is.

If $R$ is a randomized decision tree, we let $R(x)$ denote the random variable we get by picking a decision tree according to the distribution $R$ and running it on $x$. We say $R$ computes $f$ to error $\epsilon$ if $\Pr[R(x) \neq f(x)] \leq \epsilon$ for all $x$.

As for the height, note that for deterministic decision trees the height was the worst-case number of queries made. For randomized algorithms, we have two reasonable options for defining the worst-case number of queries. One option is to take the worst case of $\text{height}(D, x)$ over both decision trees $D$ in the support of $R$ and over inputs $x$. The other option is to take the worst case of $\mathbb{E}[\text{height}(R, x)]$ over inputs $x$. In the latter case, we are still considering the worst-case input, but we average out over the internal randomness of $R$ when measuring the query cost on that input. In the former case, we take the worst case over both inputs and randomness. We will denote these by $\text{height}(R)$ and $\overline{\text{height}}(R)$ respectively.

Now, how shall we define randomized query complexity $\mathrm{R}(f)$? If we use the strictest definition, requiring the algorithm to be always right and also measuring its cost in the worst case over both inputs and randomness, then the randomized query complexity becomes the same as the deterministic query complexity $\mathrm{D}(f)$. This is because at that point, any decision tree in the support of $R$ can be used to compute $f$ just as well as $R$ can.

If we relax our cost measure but not the correctness measure, we get a called zero-error randomized query complexity, denoted $\mathrm{R}_0(f)$. Zero-error randomized algorithms (also called Las Vegas algorithms) always output the right answer, and for all inputs they terminate quickly on average, but they may use a very large number of queries if you get unlucky.

Alternatively, we can relax the correctness measure but not the cost measure. This means we will allow our algorithms to be wrong with some small error $\epsilon$ (but they must still output the right answer with probability at least $1 - \epsilon$, no matter what the input is). What should the value of $\epsilon$ be? Note that at $\epsilon = 1/2$, this error can be achieved via randomly guessing $f(x)$ without reading the input at all. This means that to get a nontrivial new measure, we should pick $\epsilon \in (0, 1/2)$. As it turns out, within that range, we can move from one error level to another by *amplifying*: we can repeat a high-error algorithm several times on the same input $x$, take the majority vote of the outputs, and get an estimate for $f(x)$ whose error is smaller than the original error level. To get from one constant value of $\epsilon$ in $(0, 1/2)$ to another, we only need to increase the number of queries by a constant factor. Therefore, if we do not care about constant factors, any constant value of $\epsilon$ will do, so long as it is not 0 and not $1/2$, and so long as it is independent of $n$ and $f$. The usual convention is to take $\epsilon = 1/3$. The minimum height of a randomized decision tree computing $f$ to error $1/3$ is denoted $\mathrm{R}(f)$. $\mathrm{R}(f)$ is the minimum number of queries a bounded-error algorithm (also called Monte Carlo algorithm) must make to compute $f$ in the worst case.

Finally, what if we relax both the correctness and the cost measure? Using $\epsilon = 1/3$, this gives a measure which is sometimes written $\overline{\mathrm{R}}(f)$. The difference between $\overline{\mathrm{R}}(f)$ and $\mathrm{R}(f)$ is that in the former, the cost is the worst-case over $x$ of the *expected* number of queries $R$ makes on $x$. We have $\overline{\mathrm{R}}(f) \leq \mathrm{R}(f)$, but it turns out there is a way to convert $\overline{\mathrm{R}}(f)$ algorithms into $\mathrm{R}(f)$ algorithms. The idea is as follows: start with a $\overline{\mathrm{R}}(f)$ algorithm $R$, and run it on the input $x$. If you see it taking too long, say longer than $10\overline{\mathrm{R}}(f)$, cut it off and guess the answer randomly. This new algorithm $R'$ uses only $10\overline{\mathrm{R}}(f)$ queries in the worst case, no matter what the input or the internal randomness; hence it is an $\mathrm{R}(f)$ style algorithm. However, note that by Markov's inequality, the probability of $R$ using more than $10\overline{\mathrm{R}}(f)$ queries when its expectation is $\overline{\mathrm{R}}(f)$ is at most $1/10$. It follows that the error probability of $R'$ is at most the error probability of $R$ plus $1/10$. We can now amplify $R'$ to get an algorithm $R''$ which achieves error $1/3$ and worst-case number of queries at most $O(\overline{\mathrm{R}}(f))$, which means that $\mathrm{R}(f) = O(\overline{\mathrm{R}}(f))$. Since we don't care about constant factors, we conclude that $\overline{\mathrm{R}}(f)$

does not give us a new query complexity measure, so we have two randomized query complexities: $\mathrm{R}(f)$ and $\mathrm{R}_0(f)$.

We remark that $\mathrm{R}(f) = O(\mathrm{R}_0(f))$ (using the "cut off the algorithm and use Markov's inequality" trick above), and $\mathrm{R}_0(f) \leq \mathrm{D}(f)$.

## 1.2 Quantum query complexity

Next, we define quantum query complexity. The definition requires a bit of quantum background; however, as it turns out, we will rarely even use the definition of quantum query complexity in this course! This is because most quantum lower bounds are proven through well-established tools, and these tools themselves do not refer to the definition of quantum query complexity (or to any quantumness at all, really). We will only need the definition of quantum query complexity itself for the purpose of proving that those tools are correct—which only needs to be done once.

Still, for completeness, let us tackle this definition. Students without quantum background may find it more difficult to read through this section, but we will not rely too much on this definition for a large part of the course, so don't panic.

Before we get to the definition, it will be helpful to go over Dirac notation used for quantum states.

### 1.2.1 Dirac notation

In Dirac notation, we place each column vector $\psi$ inside a "ket", which looks like $|\psi\rangle$. This denotes that $\psi$ should be treated as a column vector. If we want a row vector, we will denote it with a "bra", which looks like $\langle\psi|$. Our vectors will be over the field of complex numbers $\mathbb{C}$, and $^\dagger$ will denote the conjugate transpose; hence we will set $\langle\psi| = |\psi\rangle^\dagger$. Note that if $|\psi\rangle$ is a column vector and $\langle\phi|$ is a row vector of the same dimension, their inner product is $\langle\phi|\psi\rangle$, a "braket". The matrix $|\psi\rangle\langle\phi|$, a "ketbra", is the outer product (a rank-1 matrix).

One of the main reasons Dirac notation is used is that it allows the following abuse of notation. If $a$ is any object, we will use the notation $|a\rangle$ to mean $|e_a\rangle$, where $e_a$ is the basis vector that has a 1 in a coordinate corresponding to $a$ and 0s elsewhere. The dimension of this vector will often remain implicit; if $a$ is an element of some set $S$, the dimension of the vector $|a\rangle$ will often be $|S|$. The way this is used is as follows. Consider a 6-sided die, which can be in any of the 1, 2, 3, 4, 5, and 6. In quantum mechanics, we will denote these states by $|1\rangle$, $|2\rangle$, $|3\rangle$, $|4\rangle$, $|5\rangle$, and $|6\rangle$. These are called the basis states. We can then also represent a die in a probabilistic state: for example, we might denote a die that has equal probability of being either in state 1 or 2 by $(|1\rangle + |2\rangle)/2$.

However, in quantum mechanics, we will often take the vectors inside braket notation to represent *amplitudes* rather than probabilities. Amplitudes are effectively square roots of probabilities, though they may be complex numbers rather than positive real numbers. To represent a die that is in equal superposition of 1 and 2, we will write $(|1\rangle + |2\rangle)/\sqrt{2}$. Note that this is a unit vector: this vector has entries $(1/\sqrt{2}, 1/\sqrt{2}, 0, 0, 0, 0)$, so its 2-norm is 1. All vectors representing quantum amplitudes must be unit vectors in some complex Hilbert space (roughly speaking, a Hilbert space is a vector space with an inner product).

Note that while this usage of Dirac notation is convenient, it can be ambiguous: if I write $|\alpha\rangle$, it is not immediately clear whether I'm referring to the column vector $\alpha$, or to the column vector $e_\alpha$ where $\alpha$ is just a basis state. Usually, variables such as $|\psi\rangle$ and $|\phi\rangle$ will be used for vectors, while variables like $|x\rangle$ and $|i\rangle$ will be used for basis states.

A second abuse of notation occurs when two kets are placed next to each other, especially when they are basis states. The notation $|x\rangle\,|i\rangle$ looks nonsensical because we cannot multiply two column

vectors together. However, we will use it to mean the Kronecker (tensor) product between the states, also denoted by $|x\rangle \otimes |i\rangle$. This is a column vector that has dimension equal to the product of the dimensions of $|x\rangle$ and $|i\rangle$; if $|x\rangle$ has one coordinate for each element of the set $S$ and $|i\rangle$ has one coordinate for each element of the set $T$, then $|x\rangle |i\rangle$ will have one coordinate for each element of the set $S \times T$. Moreover, the vector $|x\rangle |i\rangle$ will have zeros everywhere except at the entry corresponding to the pair $(x, i)$, where it will have a 1. More generally, $|\phi\rangle \otimes |\psi\rangle$ will have, at coordinate $(y, j)$, the entry $\phi_y \psi_j$.

This inner product corresponds to taking two unrelated (uncorrelated or unentangled) particles and considering them together. For example, if we have a die in probabilistic state $|p\rangle = (|1\rangle + |2\rangle + |3\rangle)/3$ and a coin in probabilistic state $|q\rangle = (|heads\rangle + |tails\rangle)/2$, and if the coin flip is independent of the die roll, then the state of the coin and die together is the product distribution of these two distributions, which, as a vector, is exactly the tensor product $|p\rangle \otimes |q\rangle$. The basis vectors of the die and coin considered together will be the vectors $|1\rangle |heads\rangle$, $|1\rangle |tails\rangle$, $|2\rangle |heads\rangle$, and so on. In quantum mechanics, the coin and die will in general be in superposition (vectors whose 2-norm is 1, rather than vectors whose 1-norm is 1), but the basis states will still be the same basis states ($|1\rangle |heads\rangle$ and so on).

### 1.2.2 Defining quantum query complexity

We wish to define quantum query algorithms, which are capable of making queries to the input $x$ in superposition. Normally, we feed in query $i$ and get as output $x_i$. However, since quantum mechanics is reversible, we will need a reversible notion of queries. We will say that a query maps the pair $(i, b)$ to the pair $(i, b \oplus x_i)$. Here $b \in \{0, 1\}$ is an arbitrary bit, which will often simply be 0, and $b \oplus x_i$ denotes xor of $b$ and $x_i$ (flipping $b$ if $x_i = 1$). The point of defining a query this way is that it can be reversed: we can get from $(i, b)$ to $(i, b \oplus x_i)$ if we know $x_i$, but we can also get from $(i, b \oplus x_i)$ back to $(i, b)$ if we know $x_i$ (in fact, making another query reverses the previous one).

The next step will be to use Dirac notation, and use $|i\rangle |b\rangle$ for the basis vector corresponding to $(i, b)$. We can then represent the map $|i\rangle |b\rangle \to |i\rangle |b \oplus x_i\rangle$ by a matrix $U^x$, which will simply be a permutation matrix, and therefore is unitary. Now, while classical algorithms are able to call this map on a single basis vector $|i\rangle |b\rangle$ at a time (possibly a randomly selected one), a quantum algorithm can call this map on a superposition of them, which is an arbitrary vector on this space whose 2-norm still equals 1.

After each query, we would like to allow the quantum algorithm to select the next query in an arbitrary way. To do so, we give the algorithm a work space of arbitrary size. That is, let $W$ be a set representing the possible basis states of the work space. Then the quantum algorithm will act on the Hilbert space with basis states corresponding to $W \times [n] \times \{0, 1\}$. The algorithm will alternate between applying an arbitrary transformation on this state which is independent of $x$ (that is, an arbitrary unitary matrix which does not depend on the input), and applying the unitary $U^x$ implementing the query specified by the query register (the unitary $U^x$ will be extended to act as identity on the work space register). That is, the action of the quantum algorithm on $x$ will be

$$U_T U^x U_{T-1} U^x U_{T-2} U^x \dots U^x U_1 U^x U_0 |\psi_{init}\rangle ,$$

where $|\psi_{init}\rangle$ is some fixed initial state. Note that $T$ is the number of queries the quantum algorithm makes, and that the quantum algorithm itself is specified by the matrices $U_0, U_1, \dots, U_T$. Note further that the unitary $U_0$ can transform the initial state to an arbitary state that does not depend on $x$; hence we can assume without loss of generality that $|\psi_{init}\rangle = |0\rangle_W |1\rangle_I |0\rangle_B$, where we assume 0 is in the basis set $W$ of the work space, and where we use the subscripts $W$, $I$, and $B$ to denote the different registers (work register, index register, and query-answer register).

Actually, we will add yet another register, which should be viewed as a special part of the work register. This new register will be the output register, and has basis states in $\{0,1\}$ if the quantum algorithm returns Boolean values. Hence we will actually set $|\psi_{init}\rangle = |0\rangle_O |0\rangle_W |1\rangle_I |0\rangle_B$. Once the algorithm terminates, the output $Q(x)$ of the algorithm $Q$ on the input $x$ will be the random variable we get by *measuring* the output register in the final state (after all the unitaries are applied). That is, we take the squared magnitude of the complex numbers in the entries of the final state $|\psi_{final}\rangle$, and then separately sum up the ones corresponding to coordinates (basis states) where the output register is 0, and to coordinates where the output register is 1. This gives us two numbers, $p_0$ and $p_1$, which depend on $Q$ and on $x$. Note that since the final vector has 2-norm equal to 1 (since we started with a norm-1 vector and since unitaries preserve the 2-norm), we must have $p_0 + p_1 = 1$. We set the random variable $Q(x)$ to be 0 with probability $p_0$ and 1 with probability $p_1$.

We have now defined a $T$-query quantum algorithm. We say that a quantum query algorithm $Q$ computes Boolean function $f$ to error $\epsilon$ if $\Pr[Q(x) \neq f(x)] \leq \epsilon$ for all inputs $x$. It turns out that quantum algorithms can be amplified just like classical ones, by repeating the algorithm several times and taking the majority vote of the outputs. For this reason, we will again pick $\epsilon = 1/3$ as the standard choice, and we will define the quantum query complexity $\mathrm{Q}(f)$ to be the minimum number $T$ such that there is a $T$-query quantum algorithm computing $f$ to error $1/3$. We will also define $\mathrm{Q}_E(f)$ to be the minimum number $T$ such that there is a $T$-query quantum algorithm computing $f$ to error 0; note that for randomized algorithms, the analogous definition gave us $\mathrm{D}(f)$, but for quantum algorithms we get a new measure $\mathrm{Q}_E(f)$. For now, we will not define expected-cost quantum algorithms, and so there will be no quantum analogue for $\mathrm{R}_0(f)$ or $\overline{\mathrm{R}}(f)$.

## 1.3 Partial functions and examples

So far, we have defined query complexity measures like $\mathrm{D}(f)$, $\mathrm{R}(f)$, and $\mathrm{Q}(f)$ for Boolean functions $f\colon \{0,1\}^n \to \{0,1\}$. As we've briefly mentioned, all these measures can also be defined on non-Boolean input and output alphabets; that is, for functions $f\colon \Sigma_I^n \to \Sigma_O$, where $\Sigma_I$ and $\Sigma_O$ are finite sets.

A different type of generalization is to *partial functions*. A partial Boolean function will be defined on a subset of $\{0,1\}^n$, instead of on all of $\{0,1\}^n$. That is, we will have some nonempty set $S \subseteq \{0,1\}^n$, and define $f\colon S \to \{0,1\}$. We will use $\mathrm{Dom}(f)$ to denote the domain $S$ of $f$.

Partial Boolean functions are not defined on all strings of length $n$. An algorithm $A$ is said to compute a partial Boolean function $f$ if $A(x)$ computes $f(x)$ for all $x \in \mathrm{Dom}(f)$ (to whatever error level we wish to consider). In other words, we do not care what the algorithm $A$ does when given as input a string $x \notin \mathrm{Dom}(f)$. All the query measures we have considered ($\mathrm{D}(f)$, $\mathrm{R}(f)$, $\mathrm{R}_0(f)$, $\mathrm{Q}(f)$, and $\mathrm{Q}_E(f)$) still make sense for partial Boolean functions. Partial Boolean functions are sometimes called promise problems, because we can view them as promising that the input $x$ will come from the set $\mathrm{Dom}(f)$ (and we do not care about the performance of the algorithm when this promise is violated). To contrast with the partial functions, regular Boolean functions are called *total* Boolean functions.

We will now go through a few examples demonstrating the differences between the various query measures. First, consider the function $f = \mathrm{OR}_n$, the OR function on $n$ bits. This is a total Boolean function, which evaluates to 1 if the input string $x$ is not all zeros. One can view this function as the "unstructured search" function: we are looking in a database of size $n$ for some element satisfying some property (the property that its bit is 1 instead of 0), and would like to know whether such an element exists.

It is easy to check that $\mathrm{D}(\mathrm{OR}_n) = n$. Why? Because in the worst case, a deterministic algorithm

will keep seeing zeros all the time. Until it finds a 1, such an algorithm cannot stop looking, because it must succeed in outputting the right answer in each input.

For a similar reason, we have $\mathrm{R}_0(\mathrm{OR}_n) = n$. To see this, recall that a zero-error algorithm must always output the correct answer. Suppose that we run such an algorithm on the all-zeros input $0^n$. In that case, it must output 0 with probability 1. Now, if it fails to read all the bits when given the input $0^n$, then it has some probability $p > 0$ of reading only a subset $S \subseteq [n]$ of the bits of the input. But now if we consider the input $x$ which has some 1 outside of $S$, we conclude that our algorithm will output 0 on $x$ with probability at least $p$ (since with probability $p$, it sees only the bits in $S$, all of which are zeros, and outputs 0). This contradicts the zero-error nature of the randomized algorithm.

Next, we have $\mathrm{R}(\mathrm{OR}_n) = \Theta(n)$, but it is not exactly $n$. This is because since a bounded error randomized algorithm is allowed to err, it can query only a random constant fraction of the input string, and output 0 if it sees all zeros and 1 if it found a 1. It is not hard to check that a randomized algorithm cannot do better than this while maintaining its bounded error. To see this, consider the distribution $\mu$ on $\{0,1\}^n$ which outputs $0^n$ with probability $1/2$, and otherwise outputs a random string of Hamming weight 1 (meaning a string with exactly one 1 in it). If a randomized algorithm $R$ solves $f$ to bounded error when given worst-case inputs, then it also solves $f$ to bounded error when given inputs from $\mu$. Now, the expected error $R$ makes against $\mu$ is equal to to the average, over $D \sim R$, of the expected error the decision tree $D$ makes against $\mu$. Since $R$ makes error at most $\epsilon$ against $\mu$, there must be some deterministic decision tree $D$ that also makes error at most $\epsilon$ against $\mu$. We know that $D(0^n) = 0$, since otherwise $D$ will make error $1/2$ against $\mu$. Now, if $S \subseteq [n]$ is the set of positions $D$ queries when given $0^n$, then $D$ outputs 0 whenever the bits in $S$ are all 0. To achieve error $\epsilon$ against $\mu$, it must be the case that the probability of $\mu$ generating a string of Hamming weight 1 whose 1 is outside of $S$ is at most $\epsilon$. In other words, we must have $(n - |S|)/2n \leq \epsilon$, or $|S| \geq n(1 - 2\epsilon)$. We conclude that the height of $D$ must be at least $n(1 - 2\epsilon)$, so $\mathrm{R}(f) \geq n(1 - 2\epsilon)$. Since we've picked $\epsilon = 1/3$, we get $\mathrm{R}(f) \geq n/3$.

Next, the quantum query complexity of $\mathrm{OR}_n$ is actually $\mathrm{Q}(\mathrm{OR}_n) = \Theta(\sqrt{n})$. The upper bound is due to Grover search, which we will not cover (at least for now). The lower bound can be proven many different ways, the first of which will likely be covered next week. Finally, we have $\mathrm{Q}_E(\mathrm{OR}_n) = n$; that is, Grover search necessarily makes error to achieve its $O(\sqrt{n})$ running time, though we won't prove this yet.

### 1.3.1 Separations for partial functions

The above example with $\mathrm{OR}_n$ did not separate the measures $\mathrm{D}(f)$, $\mathrm{R}_0(f)$, and $\mathrm{R}(f)$ from each other (or at least, not by more than a constant factor). To demonstrate the difference between these definitions, we will give some examples of partial functions which provide large separations between these measures.

First, consider the function $\textsc{GapMaj}_n$ which is defined on $n$ bits, outputs 1 if the Hamming weight of the input is at least $2n/3$, outputs 0 if the Hamming weight of the input is at most $n/3$, and has the promise that one of these two cases hold. To solve $\textsc{GapMaj}_n$, a randomized algorithm only needs to pick one bit at random and output it: if the Hamming weight of the input is large, most choices of such a bit will give 1, which is the right output for that input, and when the Hamming weight of the input is small, most such choices will give 0, which is also correct. Hence $\mathrm{R}(\textsc{GapMaj}_n) = 1$. One the other hand, a zero-error algorithm must ensure to always output the right answer; it can only stop if it has found a *proof* or *certificate* of the output of the function. In the case of $\textsc{GapMaj}_n$, such a proof requires reading more than $n/3$ bits, so $\mathrm{R}_0(\textsc{GapMaj}_n) = \Omega(n)$. We also have $\mathrm{D}(\textsc{GapMaj}_n) \geq \mathrm{R}_0(\textsc{GapMaj}_n) = \Omega(n)$. Bounded error quantum algorithms can

mimic randomized algorithms, so $\mathrm{Q}(\textsc{GapMaj}_n) = 1$, whereas we have $\mathrm{Q}_E(\textsc{GapMaj}_n) = \Omega(n)$ (we will see this later in the course).

Next, consider the function $g$ on $n$ bits whose input string is split into two parts, a left half and a right half, each of size $n/2$ or so. The promise of $g$ will be that either (a) the Hamming weight of the left half of the input string is at least $n/4$ and the Hamming weight of the right half is 0, or (b) the Hamming weight of the left half is 0 and the Hamming weight of the right half is at least $n/4$. The output of $g$ will be 0 if (a) is the case, and will be 1 if (b) is the case. Note that finding a 1 in an input $x \in \mathrm{Dom}(g)$ immediately gives a proof of the value of $g(x)$. Also, each input $x \in \mathrm{Dom}(g)$ has Hamming weight at least $n/4$. Therefore, a randomized algorithm can simply query bits of the input $x$ at random, and each such query will have probability at least $1/4$ of being a 1 and constituting a proof of the value of $g(x)$. This means that a randomized algorithm only needs to make 4 expected queries to find a proof, so $\mathrm{R}_0(g) \leq 4$. On the other hand, a deterministic algorithm will, in the worst case, keep seeing zeros no matter where it looks; it will need to examine a constant fraction of the bits before it can be guaranteed to find a 1. Hence $\mathrm{D}(g) = \Omega(n)$.

## 1.4 Relationships for total functions

It turns out that all the query measures we've defined so far are polynomially related for total functions: we have $\mathrm{D}(f) = O(\mathrm{Q}(f)^4)$ for all total functions $f$. Since $\mathrm{Q}(f)$ is the smallest query measure we've defined (it is smaller, up to constant factors, than $\mathrm{D}(f)$, $\mathrm{R}(f)$, $\mathrm{R}_0(f)$, and $\mathrm{Q}_E(f)$), and since $\mathrm{D}(f)$ is the largest query measure we've defined, this means all of them are related to each other by at most a power 4. In this section we will go over a proof that $\mathrm{D}(f) = O(\mathrm{R}_0(f)^2)$, $\mathrm{D}(f) = O(\mathrm{R}(f)^3)$, and $\mathrm{D}(f) = O(\mathrm{Q}(f)^6)$. For a somewhat old survey of such results, you can check out [BW02]. Most of the work in this section already appeared as far back as [Nis91] (and some of it earlier), but see also [BBC+01] for the relationships with quantum query complexity. A more recent survey appears in the introduction of [ABK+20].

The first tool we will need is a formal notion of certificate complexity of a Boolean function. To define certificate complexity, we will first introduce terminology for partial assignments.

**Definition 1.3** (Partial assignment)**.** *A partial assignment on $n$ bits is a string in $\{0,1,*\}^n$ representing partial knowledge of a string in $\{0,1\}^n$. If $p$ is a partial assignment, we will identify $p$ with the set $\{(i,p_i) : i \neq *\}$ of its non-$*$ entries. This will allow us to use $p \subseteq x$ to denote $p$ being consistent with an input $x \in \{0,1\}^n$. More generally, if $p$ and $q$ are two partial assignments, we say they are consistent if they agree on all their non-$*$ bits. We also use $|p|$ to denote the size of $p$, meaning the number of non-$*$ bits in $p$.*

The notion of a partial assignment will be very useful for the study of query complexity. Next, we will need the notion of a certificate.

**Definition 1.4.** *A partial assignment $p$ is called a* certificate *for $f$ if for all $x, y \in \mathrm{Dom}(f)$ with $p \subseteq x$ and $p \subseteq y$, we have $f(x) = f(y)$.*

In other words, a certificate is a partial assignment such that all inputs consistent with it have the same $f$ value. With this definition in hand, we can define the certificate complexity of an input $x$ relative to a function $f$.

**Definition 1.5.** *Let $f$ be a Boolean function, and let $x \in \mathrm{Dom}(f)$. The certificate complexity of $f$ at $x$, denoted $\mathrm{C}(f,x)$, is the minimum size of a certificate $p$ for $f$ satisfying $p \subseteq x$.*

Note that each input $x$ is itself a certificate for $f$; hence $C(f,x) \leq n$ for all $f$ and $x$. The certificate complexity of $f$ at $x$ is the minimum number of bits of $x$ that must be revealed to a skeptic in order to convince her of the value of $f(x)$. Finally, we can define the certificate complexity of the Boolean function $f$ as follows.

**Definition 1.6.** *The certificate complexity of a Boolean function $f$, denoted $\mathrm{C}(f)$, is the maximum value of $\mathrm{C}(f,x)$ over all $x \in \mathrm{Dom}(f)$. Furthermore, we define $\mathrm{C}_0(f)$ to be the maximum value of $\mathrm{C}(f,x)$ over $x \in f^{-1}(0)$, and define $\mathrm{C}_1(f)$ to be the maximum value of $\mathrm{C}(f,x)$ over $x \in f^{-1}(1)$.*

In other words, the certificate complexity of a function is the complexity of certifying the worst-case input to that function.

We observe that $\mathrm{R}_0(f) \geq \mathrm{C}(f)$. This is because a zero-error randomized algorithm must find a certificate of its input before it can output anything. More formally, we can prove this as follows: let $x$ be the input maximizing $\mathrm{C}(f,x)$, so that $\mathrm{C}(f,x) = \mathrm{C}(f)$. Let $R$ be any zero-error randomized algorithm. Suppose some decision tree $D$ in the support of $R$ makes fewer than $\mathrm{C}(f)$ queries when run on $x$. In that case, the partial assignment $p$ of bits queried by $D$ is not a certificate; this means there is some input $y$ such that $p \subseteq y$ and $f(y) \neq f(x)$. Now, $D$ will by definition output the same value on $x$ and on $y$, which means it is wrong on one of them. But in this case, $R$ errs on that input with nonzero probability, contradicting its zero-error property. We conclude that all decision trees in the support of $R$ make $\mathrm{C}(f)$ queries when run on $x$, so the expected number of queries $R$ makes on $x$ is at least $\mathrm{C}(f)$. Hence $\mathrm{R}_0(f) \geq \mathrm{C}(f)$, as desired.

Next we will show that for all total functions $f$, $\mathrm{D}(f) \leq \mathrm{C}_0(f)\,\mathrm{C}_1(f)$. This statement is not true for partial functions; it implies that $\mathrm{D}(f) \leq \mathrm{R}_0(f)^2$, which we've seen is wrong for some partial functions.

**Theorem 1.7.** *For all total Boolean functions $f$, $\mathrm{D}(f) \leq \mathrm{C}_0(f)\,\mathrm{C}_1(f)$.*

*Proof.* We describe a deterministic algorithm for computing $f$. On input $x$, the algorithm will pick an arbitrary 0-certificate $c$ with $|c| \leq \mathrm{C}_0(f)$, and query the domain of $c$ (the set of indices $i$ such that $c_i \neq *$). It will then observe some partial assignment $p \subseteq x$. If $p$ is a certificate, the algorithm can halt and output $f(x)$. If not, there is some 0-input $y$ consistent with $p$, and the algorithm will pick a 0-certificate $c \subseteq y$ such that $|c| \leq \mathrm{C}_0(f)$. It will once again query the bits in the domain of $c$, and update the partial assignment $p$ to contain all the bits seen so far.

The algorithm proceeds in this way, each time guessing a 0-certificate consistent with what's been seen so far and querying its positions, until the total bits revealed amount to their own certificate. Each round, this algorithm makes at most $\mathrm{C}_0(f)$ queries. The key observation is that in each 0-certificate $c$ must be inconsistent with each 1-certificate $d$; hence, if $f$ is total, there must be some index $i$ such that $c_i \neq *$, $d_i \neq *$, and $c_i \neq d_i$. However, this means that when we query all positions of some 0-certificate $c$, we necessarily reveal one bit of each 1-certificate. Hence for each 1-certificate $d$, so long as $d$ does not get contradicted in any round of the algorithm, it must be the case that each round reveals a new bit of $d$. After $\mathrm{C}_1(f)$ rounds, it must be the case that all bits of $d$ are revealed, or else $d$ is contradicted. This means that after $\mathrm{C}_1(f)$ rounds of the algorithm, either all 1-certificates of size at most $\mathrm{C}_1(f)$ have been contradicted, or else one of them has been revealed. In the latter case, the algorithm terminates. In the former case, the algorithm has contradicted all 1-certificates of size at most $\mathrm{C}_1(f)$; but since each 1-input contains one such small certificate, this means the algorithm contradicted all 1-inputs, which means it has found a 0-certificate. Hence in all cases, this algorithm terminates after $\mathrm{C}_1(f)$ rounds, and makes at most $\mathrm{C}_0(f)\,\mathrm{C}_1(f)$ total queries. □

This shows that $\mathrm{D}(f) \leq \mathrm{R}_0(f)^2$ for total functions. Note that the totality was used to say that each 0-certificate and each 1-certificate must contradict each other on some index $i$; for partial functions, this need not be the case.

Next, we will show that $\mathrm{D}(f) = O(\mathrm{R}(f)^3)$. To do so, we will need to define block sensitivity. First we introduce some convenient notation for manipulating strings.

**Definition 1.8.** *Let $x \in \{0,1\}^n$ and let $B \subseteq [n]$. We write $x^B$ to denote the string $x$ with the bits in $B$ flipped; that is, $(x^B)_i = x_i$ for $i \notin B$, and $(x^B)_i = 1 - x_i$ for $i \in B$. If $B = \{i\}$ for a single bit $i$, we will write $x^i$ instead of $x^{\{i\}}$.*

Next, we define the block sensitivity of an input $x$ with respect to a function $f$.

**Definition 1.9.** *Let $f$ be a Boolean function, and let $x \in \mathrm{Dom}(f)$. A sensitive block for $x$ with respect to $f$ is a set of indices $B \subseteq [n]$ such that $f(x^B) \neq f(x)$. The block sensitivity of $f$ at $x$, denoted $\mathrm{bs}(f,x)$, is the maximum number $k$ of disjoint sensitive blocks $B_1, B_2, \ldots, B_k$ that can be found for $x$ with respect to $f$; that is, it is the maximum $k$ such that there exist $B_1, B_2, \ldots, B_k \subseteq [n]$ with $B_i \cap B_j = \varnothing$ for all $i \neq j$ and such that $f(x^{B_i}) \neq f(x)$ for all $i$.*

Finally, we define the block sensitivity of $f$ as the maximum block sensitivity for any input to $f$.

**Definition 1.10.** *The block sensitivity of a Boolean function $f$, denoted $\mathrm{bs}(f)$, is the maximum value of $\mathrm{bs}(f,x)$ over all inputs $x \in \mathrm{Dom}(f)$. We also set $\mathrm{bs}_0(f)$ to be the maximum of $\mathrm{bs}(f,x)$ over all 0-inputs $x$, and set $\mathrm{bs}_1(f)$ to be the maximum of $\mathrm{bs}(f,x)$ over all 1-inputs $x$.*

Note that if $x$ has a certificate $c \subseteq x$, then any sensitive block $B$ for $x$ must overlap with $c$ (otherwise, $c$ couldn't certify the value of $f(x)$, because $c \subseteq x^B$ but $f(x^B) \neq f(x)$). This means that if there are $k$ disjoint sensitive blocks for $x$, they must overlap $k$ different bits of $c$. In particular, the maximum number of disjoint sensitive blocks for $x$ is at most the size of $c$, so $\mathrm{bs}(f,x) \leq \mathrm{C}(f,x)$ for all $f$ and $x$. This also implies that $\mathrm{bs}(f) \leq \mathrm{C}(f)$ for all $f$.

We can also define the sensitivity of a Boolean function as the maximum number of sensitive *bits* in any input—that is, sensitive blocks of size 1 each.

**Lemma 1.11.** *The sensitivity of an input $x$ with respect to a Boolean function $f$, denoted $\mathrm{s}(f,x)$, is the number of bits $i \in [n]$ such that $f(x^i) \neq f(x)$. The sensitivity of $f$, denoted $\mathrm{s}(f)$, is the maximum value of $\mathrm{s}(f,x)$ over all $x$. We define $\mathrm{s}_0(f)$ and $\mathrm{s}_1(f)$ respectively as the maximums of $\mathrm{s}(f,x)$ over 0-inputs $x$ and 1-inputs $x$ respectively.*

Note that since every sensitive bit is a sensitive block, we have $\mathrm{s}(f,x) \leq \mathrm{bs}(f,x)$ for all $x$, so $\mathrm{s}(f) \leq \mathrm{bs}(f)$.

Next, we will show that $\mathrm{C}(f) \leq \mathrm{bs}(f)\,\mathrm{s}(f) \leq \mathrm{bs}(f)^2$. This gives a polynomial relationship between $\mathrm{C}(f)$ and $\mathrm{bs}(f)$ for total functions. We will later use $\mathrm{bs}(f)$ in both upper and lower bounds for query measures such as $\mathrm{D}(f)$, $\mathrm{R}(f)$, and $\mathrm{Q}(f)$.

**Lemma 1.12.** *Let $f$ be a total Boolean function. Then $\mathrm{C}_0(f) \leq \mathrm{bs}_0(f)\,\mathrm{s}_1(f)$ and $\mathrm{C}_1(f) \leq \mathrm{bs}_1(f)\,\mathrm{s}_0(f)$. In particular, $\mathrm{C}(f) \leq \mathrm{bs}(f)\,\mathrm{s}(f) \leq \mathrm{bs}(f)^2$.*

*Proof.* Fix a 0-input $x$. Observe that if $B_1, B_2, \ldots, B_k$ is a maximal collection of disjoint sensitive blocks for $x$ with respect to $f$, then the partial assignment $p$ which reveals all the bits in $B_1 \cup B_2 \cup \cdots \cup B_k$ must be a certificate for $x$. This is because otherwise, there would be some $y$ consistent with $p$ such that $f(y) \neq f(x)$; but then the set of bits $B$ on which $x$ and $y$ disagree would be a sensitive block for $x$, and it would be disjoint from $B_1, B_2, \ldots, B_k$, which contradicts the assumption that the collection $B_1, B_2, \ldots, B_k$ is maximal.

Next, let $B_1, B_2, \ldots, B_k$ be a maximal collection of sensitive blocks, each of which is minimal; this means that each $B_i$ satisfies $f(x^{B_i}) \neq f(x)$, but for all $B \subsetneq B_i$, we have $f(x^B) = f(x)$. Let $p \subseteq x$ be the partial assignment consisting of all the positions in $B_1 \cup B_2 \cup \cdots \cup B_k$. Then $p$ is a 0-certificate consistent with $x$. The size of $p$ is at most $k$ times the maximum size of one of the blocks. Now, if $B_i$ is a minimal sensitive block for $x$, then each bit $j \in B_i$ must be sensitive for $y = x^B$. This is because $f(y^j) = f((x^B)^j) = f(x^{B \setminus \{j\}}) = f(x) \neq f(x^B) = f(y)$. Hence the size $|B_i|$ of a minimal sensitive block is at most $\mathrm{s}(f, x^{B_i}) \leq \mathrm{s}_1(f)$. We also know that $k \leq \mathrm{bs}(f, x) \leq \mathrm{bs}_0(f)$. Hence we conclude that $\mathrm{C}(f, x) \leq \mathrm{bs}(f, x)\,\mathrm{s}_1(f)$, so $\mathrm{C}_0(f) \leq \mathrm{bs}_0(f)\,\mathrm{s}_1(f)$, as desired. □

Now let's see how to use $\mathrm{bs}(f)$ to lower bound randomized algorithms. We will show that $\mathrm{R}(f) = \Omega(\mathrm{bs}(f))$. More generally, suppose we want to show that some measure $M(f)$ is at least $\mathrm{bs}(f)$. How might we do this?

Consider the input $x \in \mathrm{Dom}(f)$ that maximizes $\mathrm{bs}(f, x)$, so $\mathrm{bs}(f, x) = \mathrm{bs}(f)$. Let $k = \mathrm{bs}(f)$, and let $B_1, B_2, \ldots, B_k$ be disjoint sensitive blocks of $x$ with respect to $f$. If the measure $M$ is non-increasing when we restrict $f$ to a smaller promise set (which is the case for all sensible measures), then to prove a lower bound for $M(f)$, it suffices to lower bound $M(f')$, where $f'$ is the function $f$ restricted to the domain $S = \{x, x^{B_1}, x^{B_2}, \ldots, x^{B_k}\}$. Next, let $f''$ be the function $f'$ where we XOR each string in the input set with $x$; then the domain of $f''$ becomes $S' = \{0^n, (0^n)^{B_1}, (0^n)^{B_2}, \ldots, (0^n)^{B_k}\}$. If the measure $M$ is symmetric between the input alphabet being 0 and being 1, such an XOR applied to all strings should not change the measure $M$; hence, for sensible measure $M$, we have $M(f) \geq M(f') = M(f'')$, and it suffices to lower bound $f''$.

Next, note that all bits outside of $B_1 \cup B_2 \cup \cdots \cup B_k$ are equal to 0 in all strings in the domain of $f''$. Moreover, all the bits inside a single block $B_i$ are duplicates of each other: they take the same value in all strings of the input. If $M$ is invariant under deleting irrelevant and duplicate bits, then $M(f'') = M(f''')$, where $f'''$ is the function $\mathrm{OR}$ defined on strings of length $k$ with the promise that the Hamming weight is either 0 or 1. We denote this function by $\textsc{PromiseOR}_k$.

In conclusion, if $M$ has the reasonable properties of being non-increasing under restriction to a promise, invariant under negation of the input bits, and invariant under deleting duplicate or irrelevant bits, then $M(f) \geq M(\textsc{PromiseOR}_{\mathrm{bs}(f)})$. We note that the measures $\mathrm{R}(f)$, $\mathrm{D}(f)$, $\mathrm{R}_0(f)$, $\mathrm{Q}(f)$, and $\mathrm{Q}_E(f)$ are all well-behaved in this way, so we have $\mathrm{R}(f) \geq \mathrm{R}(\textsc{PromiseOR}_{\mathrm{bs}(f)})$, $\mathrm{Q}(f) \geq \mathrm{Q}(\textsc{PromiseOR}_{\mathrm{bs}(f)})$, and similarly for the other measures.

Previously, we've argued that $\mathrm{R}(\mathrm{OR}_n) = \Omega(n)$. In fact, our argument only used the Hamming weight 0 and 1 inputs to $\mathrm{OR}_n$, which means that we actually showed the slightly stronger statement $\mathrm{R}(\textsc{PromiseOR}_n) = \Omega(n)$. Combined with the above, this immediately gives $\mathrm{R}(f) \geq \mathrm{R}(\textsc{PromiseOR}_{\mathrm{bs}(f)}) = \Omega(\mathrm{bs}(f))$. We also have $\mathrm{Q}(\textsc{PromiseOR}_n) = \Omega(\sqrt{n})$, which implies $\mathrm{Q}(f) = \Omega(\sqrt{\mathrm{bs}(f)})$ for all $f$. However, proving this will have to wait until next week.

Note that we've already seen $\mathrm{D}(f) \leq \mathrm{C}(f)^2$ and $\mathrm{C}(f) \leq \mathrm{bs}(f)^2$, which together give $\mathrm{D}(f) \leq \mathrm{bs}(f)^4 = O(\mathrm{R}(f)^4)$. We will now improve this to $\mathrm{D}(f) \leq \mathrm{bs}(f)^3$.

**Theorem 1.13.** *For all total Boolean functions $f$, we have $\mathrm{D}(f) \leq \mathrm{C}_0(f)\,\mathrm{bs}_1(f) \leq \mathrm{bs}(f)^3$.*

*Proof.* We reanalyze the algorithm from the proof of Theorem 1.7. Recall that in that proof, we had a deterministic algorithm that worked by repeatedly guessing a 0-certificate and querying all the bits that certificate uses. That algorithm proceeds for $k$ rounds, and queries at most $\mathrm{C}_0(f)$ bits in each round. We will show that $k \leq \mathrm{bs}_1(f)$ on every input $x$ (note that $k$ may depend on the input $x$).

To see this, fix the input $x$, and consider the step right before the last round, after $k-1$ iterations. At this point, there were $k-1$ guesses for 0-certificates, and each guess was (by the way we designed the algorithm) consistent with all the previously revealed bits. Moreover, since the algorithm does

not terminate after $k-1$ rounds, we know a certificate has not yet been found, so some 1-input $y$ must exist which is consistent with all the bits seen in the first $k-1$ rounds, as well as some 0-input $z$ consistent with those same bits.

We analyze the block sensitivity of the string $y$. Note that each of the $k-1$ guesses for 0-certificates $c_1, c_2, \ldots, c_{k-1}$ were found not to lie in the input when the algorithm queried there. For each such $c_i$, define the block $B_i \subseteq [n]$ to be the set of bits where $c_i$ was found to disagree with the real input $x$. Now, since each certificate was chosen to be consistent with the previously queried bits, it must be the case that each block of disagreement $B_i$ must be disjoint with all previous blocks $B_{i-1}, B_{i-2}, \ldots, B_1$. This means all $k-1$ blocks $B_i$ are mutually disjoint. Further, note that $y$ is consistent with the real input $x$ on all bits queried in the first $k-1$ rounds, and $y$ is a 1-input; if we flip one of the blocks $B_i$ in $y$, the resulting string $y^{B_i}$ will contain the certificate $c_i$ inside it, and will therefore be a 0-input. We conclude that $B_1, B_2, \ldots, B_{k-1}$ are all disjoint sensitive blocks for $y$.

There is one additional sensitive block for $y$ which is disjoint from the rest: this is the set $B_k$ of bits on which $y$ disagrees with $z$. Since $y$ and $z$ agree on all bits queried in the first $k-1$ rounds, this block must be disjoint from the rest, and since $y^{B_k} = z$ and $f(z) = 0$, this block is sensitive for $y$. Hence $y$ has at least $k$ disjoint sensitive blocks, meaning $\mathrm{bs}(f, y) \geq k$, and hence $\mathrm{bs}_1(f) \geq k$. The total number of queries the deterministic algorithm makes is therefore at most $\mathrm{C}_0(f) \cdot k \leq \mathrm{C}_0(f)\,\mathrm{bs}_1(f)$. □

We note that this theorem implies $\mathrm{D}(f) = O(\mathrm{R}(f)^3)$ for all total functions $f$, and if you believe that $\mathrm{Q}(f) = \Omega(\sqrt{\mathrm{bs}(f)})$ (which we will show next week), it also implies that $\mathrm{D}(f) = O(\mathrm{Q}(f)^6)$. This power 6 relationship has recently been improved to a power 4 relationship [ABK+20]. Improving the power 3 relationship between $\mathrm{D}(f)$ and $\mathrm{R}(f)$ remains an open problem.

In the other direction, we know of a family of Boolean functions which has a power 4 *separation* between $\mathrm{D}(f)$ and $\mathrm{Q}(f)$ (up to log factors), so we know that the power 4 relationship is tight. On the other hand, all we know about $\mathrm{D}(f)$ vs. $\mathrm{R}(f)$ is that the maximum gap for total functions is between a power 2 and a power 3, and all we know about $\mathrm{R}(f)$ vs. $\mathrm{Q}(f)$ is that the maximum gap is between a power 3 and a power 4.

# References


[ABK+20] Scott Aaronson, Shalev Ben-David, Robin Kothari, Shravas Rao, and Avishay Tal. Degree vs. Approximate Degree and Quantum Implications of Huang's Sensitivity Theorem. Preprint, 2020. arXiv: 2010.12629 (pp. 8, 12).

[BBC+01] Robert Beals, Harry Buhrman, Richard Cleve, Michele Mosca, and Ronald De Wolf. Quantum lower bounds by polynomials. *Journal of the ACM* (2001). Previous version in FOCS 1998. DOI: 10.1145/502090.502097. arXiv: quant-ph/9802049 (p. 8).

[BW02] Harry Buhrman and Ronald de Wolf. Complexity measures and decision tree complexity: a survey. *Theoretical Computer Science* (2002). DOI: 10.1016/S0304-3975(01)00144-X (p. 8).

[Nis91] Noam Nisan. CREW PRAMs and decision trees. *SIAM Journal on Computing* (1991). Previous version in STOC 1989. DOI: 10.1137/0220062 (p. 8).