CS 514: Advanced Algorithms II – Sublinear Algorithms **Rutgers: Spring 2020**

Lecture 3

February 4, 2020

Instructor: Sepehr Assadi *Scribe: Zach Langley*

1 Lower Bounds for Sublinear Time Algorithms

In the previous lecture, we designed a sublinear time algorithm to count the number of components of a graph up to an *additive error.* We did not shoot for a multiplicative error because we claimed that even counting the number of components within a factor of two is hopeless. In this lecture, we will justify this claim by establishing the following theorem.

Theorem 1. *Let G be a graph on $n > 3$ vertices. Any algorithm (deterministic or randomized) for deciding whether or not an undirected graph G is connected in the general query model requires at least $\Omega(n^2)$ queries to output the correct answer with probability at least $2/3$.*

It may be surprising that we can make such a statement. Firstly, the claim is that something is impossible. No matter how clever one is, one cannot find a sublinear time algorithm for solving graph connectivity. Such results are often called "lower bounds" or "impossibility results" – they *lower bound* the runtime of *any* algorithm for solving a given task. Unlike NP-hardness, this lower bound is *unconditional*—no assumptions about the world like $\mathsf{P} \neq \mathsf{NP}$ need be made.

Another reason the theorem may be surprising is that unconditional lower bounds are often (extremely) hard to come by. Even SAT—a quintessentially hard problem widely conjectured to require exponential time—has eluded proof that it requires $\omega(n)$ time.[1]

The critical difference in the sublinear time regime is that a sublinear time algorithm cannot read its entire input. The fact that a sublinear time algorithm is necessarily oblivious to some of its input gives us a new avenue for proving lower bounds. For example, if no algorithm can decide graph connectivity unless it "knows about" (i.e., queries for) at least half of the edges in the graph, then we obtain a corresponding time lower bound, since every query is a step of the algorithm and takes at least constant time. Note that our goal is to prove statements purely about the amount of information in the input needed by an algorithm to compute some function on that input, namely, an *information theoretic* lower bound—the time lower bound comes as a corollary.

In order to do this, we are going to study the notion of *query complexity* (sometimes called decision tree complexity) with respect to its connection to proving sublinear time lower bounds.

Remark. Query complexity is a well studied and deep area in theoretical computer science and has various aspects and applications that go way beyond sublinear time algorithms. In this course however, we consider query complexity from a perspective of an algorithm designer and solely focus on its relation with sublinear time algorithms.

[1]That said, we do know time-space tradeoffs known for SAT, although very modest ones. For example, if an algorithm only uses $O(\sqrt{n})$ space, then $\omega(n^{1.3})$ time is required. More generally, any algorithm that uses $S(n)$ space and $T(n)$ time with $S(n)T(n) = O(n^{1.801})$ cannot solve SAT; see [2].

2 Deterministic Query Complexity

In the query complexity model, a function $f : \{0,1\}^n \to \{0,1\}$ is computed on an input x by making a sequence of *queries* to the input x. A query asks for the i-th bit of the input x. For example, if $x = 0010$, then querying $i = 1$ would reveal $x_1 = 0$ and querying $i = 3$ would reveal $x_3 = 1$. The goal is to compute f with as few queries as possible. Note that there is no notion of time in this model; algorithms are only charged for the queries made to the input string and otherwise have unlimited computational power.

Every deterministic algorithm for computing a function in this model corresponds to a decision tree. A *decision tree* is a rooted binary tree where each node is labeled by one of the inputs. If the input x_i is a 0, computation proceeds to the left-child, and if the input x_i is a 1, computation proceeds to the right-child. Finally, all leaves of the decision tree are labeled with either 0 or 1, corresponding to the output. See Figure 1 for an example.

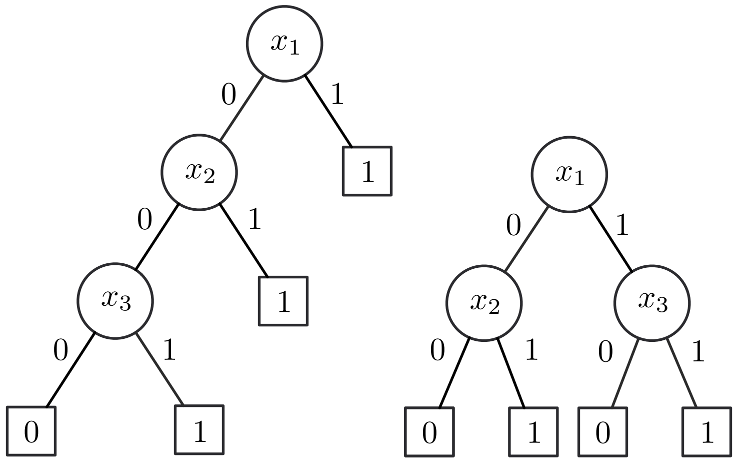

Figure 1: A decision tree for the function OR_3 defined by $(x_1, x_2, x_3) \mapsto x_1 \vee x_2 \vee x_3$ (left) and one for the addressing function defined by $f(x_1, x_2, x_3) = (\bar{x}_1 \wedge x_2) \vee (x_1 \wedge x_2)$, which outputs x_2 if $x_1 = 0$ and x_3 if $x_1 = 1$. (right).

Definition 2. The *deterministic query complexity* $D(f)$ of a function $f : \{0,1\}^n \to \{0,1\}$ is the worst-case number of queries made in the by the best algorithm computing f, namely, the algorithm that minimizes the number of queries in the worst case.

Equivalently, $D(f)$ is the minimum depth over all decision tree computing f, minus one (we do not want to count the leaf node). Trivially, we have $D(f) \geq 1$ for all non-constant functions f and $D(f) \leq n$ for all functions f.

2.1 Promise Problems

We will also consider computing functions not defined on the entire domain $\{0,1\}^n$.

For example the function:

$$M_n(x_1, \ldots, x_n) := \begin{cases} 0 & \text{if } |x| \leq n/3, \\ 1 & \text{if } |x| \geq 2n/3. \end{cases}$$

is undefined depending on $|x|$ (here $|x|$ denotes the number of ones in x, i.e., the *Hamming weight* of x). However, we can still talk about the query complexity of an algorithm for computing M_n, provided that the algorithm only needs to compute the correct output when $|x| \notin (n/3, 2n/3)$ (otherwise, the algorithm may output an arbitrary value).

Promise problems are often used to relax functions that may be hard to compute. For example, if we are promised that the number of components in a graph is either more than $n/10$ or less than $n/5$, then there is a sublinear time algorithm deciding in which class the input graph lies, as we saw in the previous lecture. The function M_n above may be seen as a relaxation of the majority function MAJ_n that outputs 1 if a majority of the input bits are 1 and 0 otherwise.

Remark. One can also consider query complexity for relations, namely, when some inputs can map to several valid outputs, or when the range of the function is non-binary, and many other variants.

2.2 Adversary Arguments

Figure 1 shows a decision tree for OR_3 of depth three. Can we do better? I.e., is there a deterministic algorithm that computes OR_n with fewer than n queries? Intuitively, the answer seems to be no. Consider inputs that have exactly one 1 in them. If a deterministic algorithm fails to find the only bit set, it cannot output the correct value. Let us formalize this intuition.

Proposition 3. *The deterministic query complexity of* OR_n *is* $D(\mathrm{OR}_n) = n$.

It is not hard to prove this theorem from first principles, but let us develop a useful proof technique known as an *adversary argument* for this purpose.

Adversary arguments: In an adversary argument, we imagine a two-player game between the Algorithm and an Adversary based on some Boolean function f. The game is played as follows.

1. The Adversary maintains a bag of strings S, initialized to contain the entire function domain. For a standard function, $S = \{0,1\}^n$. For a promise problem, the initial set would contain the subset of strings that satisfy the promise.

2. In each round of the game, the Algorithm is allowed to query a new bit, say, the i-th bit, and the Adversary answers with $x_i \in \{y_i : \exists y \in S\}$. In words, the Adversary must choose an output that actually describes an element in S. The set S is updated to remove all elements y such that $y_i \neq x_i$.

3. The game ends if $f(y)$ takes the same value for all $y \in S$.

We claim that the length of any such game with respect to f is a lower bound on $D(f)$. Here is an informal argument. Fix a decision tree for f. The Algorithm in the game simply asks the Adversary queries following the evaluation of the decision tree, using the Adversary's responses to navigate down the tree. The bag S corresponds to all strings that would have lead to the Algorithms current node, and so the game ends if all leaf nodes under the current node have the same label (because the algorithm can now output the value of f and needs no further queries). Thus, as long as the game continues, there are at least two leaf nodes descending from the current node with different labels. The game proceeding to k rounds, therefore, implies that the depth of the decision tree is at least k.

To apply an adversary argument, we fix some algorithm for the input and design an adversary that prolongs the game as much as possible so obtain stronger lower bounds.

Proof of Proposition 3. Fix an algorithm (decision tree) for OR_n. Consider the adversary that always returns every query with 0. After any $n-1$ queries, there is some index $i \in [n]$ not queried yet; suppose without loss of generality that $i = 1$. At this point, the Adversary's bag contains at least two strings evaluating to different values, namely 0^n and 10^{n-1}. Thus, the game proceeds to the n-th round, giving $D(\mathrm{OR}_n) \geq n$. □

We can also use an adversary argument to prove a lower bound for the promise function M_n defined in the previous section.

Proposition 4. *The function M_n has deterministic query complexity at least $2n/3 + 1$.*

Proof. Assume for simplicity that n is a multiple of 3. Consider the adversary that outputs $n/3$ zeros and then $2n/3$ ones. After $2n/3$ rounds, the adversary will have revealed $n/3$ zeros and $n/3$ ones. Setting the unqueried indices $n/3$ indices to 1 produces a string in S evaluating to 1 and setting the remaining $n/3$ indices to 0 produces a string S evaluating to 0. $\square$

Remark. Adversary arguments form one of the key techniques for proving deterministic query complexity lower bounds and we will use them throughout the course whenever we want to prove such bounds. However, they are by no means the only technique for proving query complexity lower bounds and various tools and techniques have been developed for this purpose. See, e.g. [1] for a survey of some of these classical tools.

3 Randomized Query Complexity

We now turn to randomized algorithms. We can think of randomized algorithms in two ways. A common view is that a randomized algorithm has access to a random stream of bits, and pulls from this stream as it computes. Alternatively, we may view a randomized algorithm as a distribution over deterministic algorithms (or, in our case, decision trees). To understand the latter interpretation, note that we can "instantiate" the random stream of bits before the algorithm beginss, after which point the algorithm becomes deterministic.

For a randomized algorithm ALG, let ALG_r be the deterministic algorithm by fixing the randomness over ALG to be the string r, and let $\mathrm{ALG}_r(x)$ denote the output of ALG_r on input x. For concreteness, you may think of r as an infinite string living in $\{0,1\}^{\mathbb{N}}$. A randomized query algorithm ALG computes a function $f : S \to \{0,1\}$ if for all $x \in S$, it holds that

$$\Pr_r(f(x) = \mathrm{ALG}_r(x)) \geq 2/3,$$

where r ranges over all possible random bit sequences queried by ALG. As usual, the constant $2/3$ is arbitrary; anything bounded away from $1/2$ sufficiently (say, by a constant) will be equivalent, ignoring constant factors.

Definition 5. The *randomized query complexity* $R(f)$ of a function f is the worst-case number of queries made by the best randomized ALG computing f (over any input and any random stream).

We should emphasize that the definition above is about the worst-case number of queries—not the expected number of queries.

Determinism vs Randomization: With respect to query complexity, randomized algorithms are (much) more powerful than deterministic algorithms. Consider the following randomized algorithm for computing the promise function M_n introduced earlier with just one query:

1. Pick an $i \in [n]$ uniformly at random.
2. Output x_i.

We consider two cases to show that the algorithm works. If $|x| \geq 2n/3$, then the algorithm samples and outputs 1 and therefore succeeds with probability $2/3$. Otherwise, if $|x| \leq 1/3$, then the algorithm samples and outputs 0 and therefore succeeds with probability $2/3$. Thus, while $D(M_n) > 2n/3$, we have $R(M_n) = 1$.

As an aside, the alert reader may have noticed the happy coincidence that the definition of M_n uses the same fraction as the probability of success needed by a randomized algorithm. If the constants were changed in either definition, we may no longer have $R(M_n) = 1$, but we would still have $R(M_n) = O(1)$. This follows from the usual amplification strategy: run C trials and take the median (i.e., majority, when dealing with binary values) output. The number of trials C depends on only the constants in the two definitions and is therefore constant with respect to the input length.

Lower Bounds for Randomized Algorithms? As already observed in the example above, randomized algorithms can be much more powerful than deterministic ones which makes proving lower bounds for them a more challenging task. For instance, the type of adversary arguments appealed to in the previous section rely heavily on the determinism of the algorithm and no longer apply to randomized algorithms (as is apparent from the upper bound of $R(M_n) = O(1)$ even though there is a "long" adversary-algorithm game for this problem). As such, to prove lower bounds for randomized algorithms, we are going to explore new ideas.

3.1 Distributional Complexity

Let μ be a distribution over $\{0,1\}^n$. We say that a deterministic algorithm *computes f over μ* if the probability that the algorithm errs on $x \sim \mu$ is at most $1/3$ (we emphasize that the randomness here is over the choice of input and the algorithm is deterministic again).

Definition 6. The *distributional complexity* $D_\mu(f)$ of a function f is the worst-case number of queries made by the best algorithm computing f over μ with probability of success at least $2/3$ (over the randomness of the distribution).

Note that we have switched from talking about distributions over deterministic algorithms (randomized algorithms) to distributions over inputs. It turns out however that these two measures are intimately connected to each other. In particular, the celebrated *Yao's minimax principle* [3] relates distributional query complexity to randomized query complexity as follows.

Proposition 7 (Yao's minimax principle)**.** *For any function $f : \{0,1\}^n \to \{0,1\}$,*

(a) $D_\mu(f) \leq R(f)$ for every distribution μ over $\{0,1\}^n$, and
(b) $D_\mu(f) = R(f)$ for some such distribution μ.

In other words, (a) says that if we want to lower bound $R(f)$, it suffices to lower bound $D_\mu(f)$ for a μ of our choosing (which brings us back to proving lower bounds for deterministic algorithms albeit on a distribution). Part (b) says that—provided we are clever enough—we can find a μ such that $D_\mu(f)$ gives us $R(f)$ exactly[2]. The important piece for us is part (a) as it gives us the means to lower bound $R(f)$.

Since we only need part (a), we only prove part (a) in this lecture.

Proof of Proposition 7-(a). The proof of this result is simply an averaging argument, which we will outline below in details.

Let ALG be a randomized algorithm for f achieving query complexity $R(f)$. For any choice of random string r, let ALG_r denote the deterministic algorithm obtained from ALG after fixing its randomness to be r. Define the indicator random variable $Z(x,r)$ for $x \sim \mu$ and random bits r to denote the event that $\text{ALG}_r(x) = f(x)$, namely, ALG outputs the correct answer on input x conditioned on its random bits being r. On one hand, we have,

$$\begin{aligned}
\mathop{\mathbb{E}}_{r,x\sim\mu} Z(x,r) &= \sum_{x,r} \Pr(x) \cdot \Pr(r \mid x) \cdot Z(x,r) && \text{(by law of conditional expectations)} \\
&= \sum_{x,r} \Pr(x) \cdot \Pr(r) \cdot Z(x,r) \\
& \qquad \text{(as } r \perp x \text{, i.e., } r \text{ is independent of } x \text{ (by definition of a randomized algorithm))} \\
&= \sum_{x} \Pr(x) \cdot \mathop{\mathbb{E}}_{r} [Z(x,r)] \geq \sum_{x} \Pr(x) \cdot \frac{2}{3} = \frac{2}{3},
\end{aligned}$$

where the last inequality is because for any fixed x, ALG outputs the correct answer with probability at

[2] This is only one interpretation of this result and in some scenarios one may want to view this differently.

least 2/3 (by definition of a randomized algorithm). On the other hand,

$$\mathop{\mathbb{E}}_{r,x\sim\mu} Z(x,r) = \sum_{x,r} \Pr(x)\cdot\Pr(r)\cdot Z(x,r) \qquad \text{(by second equality above)}$$
$$= \sum_{r} \Pr(r)\cdot \mathop{\mathbb{E}}_{x\sim\mu} Z(x,r) \leq \max_{r} \mathop{\mathbb{E}}_{x\sim\mu}\left[Z(x,r)\right].$$
(as maximum can only be larger than expectation)

This means that there is a fixed choice of r, where $\mathbb{E}_{x\sim\mu}[Z(x,r)] \geq 2/3$; in words, this means that there is a choice of r where the deterministic algorithm ALG_r outputs the correct answer with probability 2/3 over choices of $x\sim\mu$. But since query complexity of ALG_r is at most of that of ALG (as we consider worst-case query complexity), we obtain that $D_\mu(f) \leq R(f)$. □

Remark. Part (a) of Proposition 7 is often called the *easy direction* of Yao's minimax principle or simply an averaging argument.

Important note: When working with distributional query complexity and in particular in the context of Yao's minimax principle, remember that the deterministic algorithm ALG is chosen *after* fixing the distribution of inputs μ, i.e., ALG is a function of the distribution.

We now use the easy direction of Yao's minimax principle to prove a randomized query complexity for OR.

Proposition 8. *The randomized query complexity of* OR_n *is* $R(\text{OR}_n) \geq n/3$.

Proof. We only need to establish the lower bound $D_\mu(\text{OR}_n) \geq n/3$ for some distribution μ of our own choosing. By Yao's minimax principle, it will follow that $R(\text{OR}_n) \geq n/3$ as well.

We propose the following distribution of inputs $x \in \{0,1\}^n$:

(i) Sample $i \in [n]$ and $\theta \in \{0,1\}$ uniformly at random and independently.

(ii) Set $x_i = \theta$ and $x_j = 0$ for $j \neq i$.

Fix any deterministic algorithm ALG computing OR_n on distribution μ with probability at least 2/3. We use q to denote the number of queries made by ALG. Let $K = \{i_1,\ldots,i_q\}$ be the set of queries asked by ALG assuming the answer to every one of these queries were 0 (figuratively, these are queries in the "left most branch" of the decision tree). Since ALG is deterministic, this set is always fixed (and well-defined). Additionally, for any x, let $Q(x)$ denote the set of indices queried by the algorithm when run on the input x. Since ALG is deterministic, it follows that for any $x\sim\mu$, whenever $i \notin K$, we have $Q(x) = K$. Moreover,

$$\Pr(i \in K) = \frac{|K|}{n} = \frac{q}{n}.$$

At the same time, conditioned on $i \notin K$, $x_i = \theta$ is *not* queried and according to the distribution, x_i is still chosen from $\{0,1\}$ uniformly at random. But the answer of the algorithm is now a deterministic function of $Q(x) = K$ and hence is either fixed to 0 or 1. Hence,

$$\Pr(\text{ALG outputs correct answer} \mid i \notin K) = \frac{1}{2}.$$

Define $E(x)$ as the event that ALG succeeds on computing x. By combining the above, we have,

$$\frac{2}{3} \leq \Pr_{x\sim\mu}(E(x)) = \Pr(i\in K)\cdot\Pr(E(x)\mid i\in K) + \Pr(i\notin K)\Pr(E(x)\mid i\notin K)$$
$$\leq \frac{q}{n} + \left(1-\frac{q}{n}\right)\cdot\frac{1}{2}, \qquad \text{(as } \Pr(E(x)\mid i\in K)\leq 1\text{)}$$

which implies (by a simple calculation) $q \geq n/3$. Consequently, $D_\mu(OR_n) \geq n/3$ as well and in turn $R(OR_n) \geq n/3$, proving the lower bound. $\square$

Important note: Note that in Proposition 8, we proved the stronger statement that even if we are promised that in the input string to OR_n, there is at most one 1, a randomized algorithm still needs to make $n/3$ queries before solving OR_n.

Remark. One can show with a more clever distribution and analysis that $R(\mathrm{OR}_n) \geq n/2$. Similarly, a clever randomized algorithm shows $R(\mathrm{OR}_n) \leq n/2$. Thus, $R(\mathrm{OR}_n) = n/2$. However, for our purposes, these details are not important, and so we leave them to the interested reader.

4 Back to Graph Connectivity: Proof of Theorem 1

We have now developed sufficient theory to prove the graph connectivity lower bound we mentioned in Theorem 1 at the beginning of these notes. The main idea now is show that any algorithm for graph connectivity in the general query model taking q queries implies an algorithm for computing OR in some $f(q)$ queries. Since we know a query lower bound for computing OR, we inherit a query lower bound for computing graph connectivity via a *reduction*. In other words, we will describe a reduction from OR to graph connectivity.

Proof of Theorem 1. Let $N = \binom{n}{2}$. We will reduce from OR_N when promised that there is at most one input bit set to one. The reduction consists of two parts. First, we must define a mapping from an $x \in \{0,1\}^N$ to a graph G_x such that $\mathrm{OR}_n(x) = 1$ if and only if G_x is connected. Second, we must show that we can simulate all graph queries in the general query model by making at most one query to the pre-image.

We define the mapping from an $x \in \{0,1\}^N$ to a graph G_x on $2n$ vertices as follows. Let $V(G_x) = U \cup V$, where $U = \{u_1, \ldots, u_n\}$ and $V = \{v_1, \ldots, v_n\}$, and $U \cap V = \emptyset$. For notational simplicity, we assume that x is indexed by pairs i, j from $[n]$ where $i < j$. If $x_{ij} = 1$, then G_x contains the "cross edges" $u_i v_j$ and $u_j v_i$. If $x_{ij} = 0$, then G_x contains the "internal edges" $u_i u_j$ and $u_j v_j$. No other edges are added to G_x. See Figure 2 for an example.

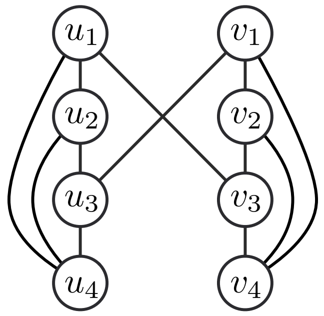

Figure 2: The graph G_x corresponding to the string $x = 010000$. The bits are indexed by the vertex pairs $(12, 13, 14, 23, 24, 34)$.

If $x = 0^n$, clearly the graph G_x is disconnected, as there are no edges between U and V. Otherwise, U is a clique minus an edge, V is a clique minus an edge, and there is an edge between U and V, and so G_x is connected provided that $n \geq 3$.

Now we must show that we can simulate the general query model on G_x with queries to x. There are three types of queries:

1. *Degree queries.* For degree queries, we can simply return $n - 1$. Indeed, every vertex u_i is connected to exactly one vertex in $\{u_j, v_j\}$ for each $j \neq i$, and these comprise all of its neighbors.

2. *Neighbor queries.* We can assume we are given a vertex u_i. If we are returning the k-th neighbor, we set $j \leftarrow k$ if $k < i$ and $j \leftarrow k+1$ otherwise. Then x_{ij} tells whether to return v_j or u_j.

3. *Pair queries.* If $u_i v_i$ is queried, we return 0. Otherwise, we are emulating a query $u_i v_j$ for some $i < j$ and we return x_{ij}.

The reduction implies that any algorithm deciding the connectivity of a graph on $2n$ vertices with fewer than $N/3$ queries would also decide OR_N (on inputs with at most one 1) with fewer than $N/3$ queries, a contradiction with our result in Proposition 8. Thus, deciding connectivity in a graph on $2n$ vertices requires $N/3 = \binom{n}{2}/3 = \Omega(n^2)$ queries.

□

A Remark on Sublinear Time Reductions: The principle behind the reduction in Theorem 1 is the same as any other reduction (say, NP-hardness reductions) you have encountered before. Given a problem A which we already know is "hard", we prove that another problem B is also hard by showing that any algorithm for B, in a *black-box* way, will solve any input of problem A as well. This is done by "transforming" the input x of problem A into an input y for problem B and then run the algorithm for B to compute $B(y)$ and use the answer to decide $A(x)$ as well.

So what is different here? In the classical setting, the reduction has enough time to read the input x entirely and then transform it into an input y for B; for a sublinear time algorithm, this is impossible. As such, we should construct the input y of B "on the fly" or in a "need-to-know basis": this means that whenever the algorithm for B needs to know a specific part of y, we would generate that part just then and answer the algorithm for B accordingly. More formally, we answer the queries to input y by querying input x instead and then transforming the answer. This is the reason in the proof of Theorem 1, we also needed to provide a way of answering queries of to y based on queries to x. This ensures that if the runtime or query complexity of the algorithm for B is "small", then through the reduction we do not query x by much either and thus get an algorithm with small query complexity for solving A on x, which would contradicts the lower bound we know for A.

References

[1] Harry Buhrman and Ronald de Wolf. Complexity measures and decision tree complexity: a survey. *Theor. Comput. Sci.*, 288(1):21–43, 2002. 4

[2] R. Ryan Williams. Time-space tradeoffs for counting NP solutions modulo integers. *Comput. Complexity*, 17(2):179–219, 2008. 1

[3] Andrew Chi-Chih Yao. Probabilistic computations: Toward a unified measure of complexity (extended abstract). In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 222–227, 1977. 5