In this course we will learn how to prove lower bounds by designing algorithms. That is, to show that no efficient procedure can solve problem X, we design an efficient algorithm that solves problem Y. This approach, often called "the algorithmic method", was pioneered by Williams starting in [Wil10], led to successes where other approaches failed, and is currently the most popular approach.

The term **"the algorithmic method"** often refers to one specific approach (i.e., the one in [Wil10] and in direct follow-ups), whereas various different methods of proving lower bounds by designing algorithms have been developed since and are used today. I'll call those **"algorithmic approaches"** in plural.

In contrast to approaches to lower bounds that have a "low-level combinatorial" flavor and that were popular in the 1980's and 1990's, recent algorithmic approaches combine combinatorics, algorithm design, and complexity-theoretic tools (e.g., hierarchy results, proof systems, win-win arguments, and so on).

Does this make sense? There can be conflicting intuitions about the idea of using algorithms to prove lower bounds:

1. This seems odd, because why would designing an algorithm for some problem lead us to deduce that there is *no* algorithm for *another* problem?

2. This seems inherent, because proving lower bounds is always an algorithmic problem: A random function is hard, and we are looking for an *explicit* hard function, i.e. an algorithmic upper bound on computing a hard function.

3. This seems like it would be excellent news, because (supposedly) designing algorithms tends to be easier for people than proving lower bounds.

The various proofs that we'll see match different intuitions. For example, certain proofs will follow the second intuition, casting the problem of finding a hard function as an algorithmic problem, and designing an algorithm that solves it. Other proofs will match the first intuition, following a convoluted path between the algorithmic problem being solved and the lower bound being deduced (i.e., the connection between the two is opaque). Lastly, matching the third intuition, some proofs will show that even solving seemingly easy algorithmic problems is enough to deduce lower bounds.

1 Non-uniform circuits

The focus of this course will be proving lower bounds against a computational model that is stronger than TMs. This model has been the focus of much attention in TCS, and in particular one of the main attempts at making progress towards $\mathcal{P} \neq \mathcal{NP}$ is through this model. We will explain why after defining the model.

1.1 Model definition and motivation

The following model of Boolean circuits will be the one we will use most often, and we think of it as the standard model of general Boolean circuits.

Definition 1. *A* Boolean circuit *is a DAG with in-degree ("fan-in") at most two, where each gate is labeled by one of the following:*

1. **(Input gates.)** *An integer specifying an index of an input variable. In this case the gate has fan-in zero.*
2. **(Intermediary gates.)** *A unary or binary Boolean function (depending on the gate's fan-in).*
3. **(Output gates.)** *A Boolean function (like intermediary gates), and in addition an integer specifying an index of an output variable. In this case the out-degree ("fan-out") is zero.*

A circuit computes a Boolean function is the obvious way, propagating values of the inputs through the DAG. The main complexity parameter is the circuit size, which is defined as the number of gates.

The type of allowed Boolean functions is called the basis. For concreteness, we allow $\wedge, \vee, \neg$ labels. The choice of basis does not matter that much: as long as we don't mind a multiplicative constant overhead in size, we can use any complete basis (e.g., only NAND gates) and simulate any binary function by a constant-sized gadget.

Later on we will be concerned with more refined variations, each with its own set of definitional issues. For example, we might restrict the circuit's depth; in this case we may need to allow larger fan-ins (otherwise an output gate may depend on too few input gates), and hence the choice of basis will matter. Another example is counting the size as the number of wires, which is a more fine-grained measure.[1] For now let us stay focused on the standard model of general circuits.

Asymptotics and complexity classes. As usual, we are concerned about asymptotics, i.e. dependency of size on the input length. So we will define circuit families, where for every input length $n \in \mathbb{N}$ there is a circuit C_n with n input gates.

It's instructive to compare this to TMs: given a TM M, we can define the function that M computes on n-bit inputs as M_n and think of the family of functions $\{M_n\}$ defined by M. The crucial difference is non-uniformity: for $\{M_n\}$, the entire family is specified by a single M with a finite description (what we usually think of as "constant"). In contrast, a circuit family might not have any finite description!

Definition 2. *The class of languages decidable by circuits of size $s(n)$ is*

$$\mathcal{SIZE}[s] = \Big\{ L \subseteq \{0,1\}^* : \exists C = \{C_n : \{0,1\}^n \to \{0,1\}\}, \forall n \in \mathbb{N}, \mathit{size}(C_n) \leq s(n) \\ \forall x \in \{0,1\}^*, \ C_{|x|}(x) = 1 \iff x \in L \Big\}$$

[1] The number of wires may range between linear and quadratic in the number of gates.

Non-triviality. Non-uniformity cannot be waved away wlog, it actually strengthens the computational model. Here are two indications:

Exercise 3. *Show that* $\mathcal{SIZE}[2^n] = 2^{\{0,1\}^*}$. *(We will see an improvement later.)*

Exercise 4. *Show that* $\mathcal{SIZE}[1]$ *contains undecidable languages.*

These two indications are somewhat abnormal (arguably even pathological), and one might wonder if non-uniformity gives additional computational power for solving natural problems. The question of how much power non-uniformity actually gives is a major open question that we'll round back to again and again throughout the course.

For completeness, here are two examples in which circuits give more power than TMs are known to have, but there is no matching lower bound for TMs.

Theorem 5 ([Sip83; Lau83; BF99]). $\mathcal{DTIME}[T] \subseteq \mathcal{SIZE}[O(n \cdot T)]$.

Theorem 6 ([Uhl74; Uhl92]). *Let* $L \in \mathcal{DTIME}[T]$*, and consider the direct-product problem* $L^{\times k}$ *in which the input is* $(x_1, ..., x_k)$ *(each of the same length) and the required output is* $L(x_1), ..., L(x_k)$*. Then, for some* L, T, k *we have* $L^{\times k} \in \mathcal{SIZE}[o(k \cdot T)]$.[2]

The first task might also be doable by uniform TMs, and in fact this follows from strong enough hardness assumptions. For the second task, it seems reasonable to conjecture that for some L, T, k uniform machines need time $k \cdot T$ to compute $L^{\times k}$; this is known as a strong direct-product conjecture.

Why study circuit complexity? If one believes that TMs are the right model that captures general computation, then non-uniform circuits clearly have overly strong features. In this case, studying this model requires justification.

- Arguably a cleaner and more elegant model, compared to TMs (which are often thought of as quite messy to work with). Moreover, it seems a-priori very suited to using graph-theoretic techniques (e.g., combinatorics, spectral analysis, etc.). The rationale here, as is often the case in math, is to study a question that might be formally harder, but it is cleaner and easier to think of.
- It has advantages as a natural model. First, it can model electrical circuits, neural networks, and various other circuits used in practice; this depends on a suitable choice of gates.[3] Secondly, studying circuits allows more easily to speak of exact complexity; that is, after fixing a basis, we can talk of the precise number of gates required to compute a function, whereas in TMs we need to fix many more model-dependent choices (e.g., the alphabet and the number of tapes).

[2] In fact, as long as $k = 2^{o(n/\log n)}$, for *every* L, T we can compute $L^{\times k}$ in size $O(2^n/n)$.

[3] That is, either one allows more general gates, possibly with non-Boolean values (which is indeed a fruitful line of study), or one considers binary Boolean gates and thinks of this as a simplified model.

- Lower bounds against non-uniform circuits turn out to be necessary for many important lower bounds against TMs. This phenomenon is usually referred to as "Karp-Lipton theorems", named after the original theorem.

  For example, [KL82] showed that $\Sigma_2 \neq \Pi_2 \Rightarrow \mathcal{NP} \not\subset \cup_c \mathcal{SIZE}[n^c]$. The meaning is that if we want to separate $\mathcal{P}$ from $\mathcal{NP}$ in a "robust" way (i.e., also separate the second level of the polynomial-time hierarchy rather than the first), we must show that $\mathcal{NP}$ is hard "non-uniform $\mathcal{P}$".[4] Another example is $\mathcal{MA} \neq \mathcal{EXP} \Rightarrow \mathcal{EXP} \not\subset \cup_c \mathcal{SIZE}[n^c]$, proved in [BFN+93]: Separating "randomized $\mathcal{NP}$" from $\mathcal{EXP}$ (which seems like a modest goal) requires circuit lower bounds.

  In fact, it's not only lower bounds against TMs that imply circuit lower bounds – many algorithmic results (e.g., mildly improving on the best-known fine-grained runtimes for solving certain problems) also turn out to imply circuit lower bounds.

- Circuits can model TMs equipped with worst-case auxiliary information. That is, when we think of an efficient adversary holding an unknown auxiliary input z, it is convenient to model the adversary as a non-uniform circuit that has z "hard-wired". This is frequently done in pseudorandomness and in cryptography.

Also, we *can, in fact*, prove many interesting circuit lower bounds. (This is not a motivation to study them, of course, but it counters the demotivating argument that circuit lower bounds are unattainable.) This is important to note due to unjustified attitudes that treat circuit lower bounds as so hard to prove that they are supposedly unattainable, and form a "barrier" for progress.

1.2 Basic results

The following result is a sanity check, confirming that non-uniform circuits are indeed at least as strong as Turing machines.[5]

Theorem 7. *For every $T\colon \mathbb{N} \to \mathbb{N}$,*

$$\mathcal{DTIME}[T] \subset \mathcal{SIZE}[\tilde{O}(T)] \,.$$

Proof. For simplicity, we prove the weaker result $\mathcal{DTIME}[T] \subset \mathcal{SIZE}[O(T^2)]$, and only for the model of one-tape Turing machines. For every $L \in \mathcal{DTIME}[T]$ decided by T-time machine M, and every $n \in \mathbb{N}$, we create a circuit C_n of size $O(T(n)^2)$ that decides $L_n = L \cap \{0,1\}^n$, as follows.

Consider the execution of M on an input of length n as a sequence of instantaneous configurations. Each configuration contains the contents of the worktape, the location

[4] In fact, even $\mathcal{P} = \mathcal{NP}$ requires circuit lower bounds, in particular $\mathcal{E} \not\subset \mathcal{SIZE}[\Omega(2^n/n)]$. The only world in which we avoid both this circuit lower bound and the circuit lower bound $\mathcal{NP} \not\subset \cup_c \mathcal{SIZE}[n^c]$ is if $\mathcal{P} \neq \mathcal{NP}$ but $\Sigma_2 = \pi_2$.

[5] Being more accurate, there is a logarithmic overhead when simulating TMs by circuits, and we don't know if this overhead is necessary. For context, recall that we also don't know how to simulate k-tape TMs by 2-tape TMs without such overhead.

of the head, and the state of the FSM. We represent each configuration using a string of length $O(T)$, which contains T blocks each with $O(1)$ bits; in each block i we write the contents of the i^{th} cell on the worktape, and additionally whether or not the head is in the i^{th} cell (and if it is, what is the state of the FSM).

This sequence can thus be represented as a $(T+1) \times O(T)$ matrix, with the first row representing the initial state with the input written on the worktape.

The key observation is that this sequence is "downward self-reducible" in a very local way. Specifically, in each row r, each block i only depends on blocks $i-1, i, i+1$ in row $r-1$. (This is because at each step, the Turing machine can only modify the cell on which the head resides, and move the head one cell to the right or left.) This dependency naturally lends itself to a circuit C_n:

- The first layer of C_n contains n input gates.
- There are T representative layers, each with $O(T)$ gates, such that in representative layer $r \in [T]$, the values of the gates when C_n is given input x represent the configuration of $M(x)$ in the r^{th} step.
- Between each pair of representative layers we add circuitry that computes the gate values in each block as a function of the gate values in the three blocks below it. Since each block has constantly many bits, we can compute this function by brute-force; we only add constantly many layers between each pair of representative layers, and constantly many gates per block.

The size of C_n is thus $O(T(n)^2)$, and its correctness follows by simple induction. ■

Note that the proof above also extends to solving search problems, without any additional overhead. Optimizing $O(T^2)$ to $\tilde{O}(T)$ is done by first simulating the TM by an oblivious TM, whose sequence of head movements are known in advance (i.e., do not depend on the input, but only on its length).

The following claim is another sanity check, showing that most functions are hard for small circuits. The meaning is that circuits are not so strong that they can easily compute every function. As usual, the problem is finding an *explicit* hard function.

Theorem 8. *When choosing a function $f\colon \{0,1\}^n \to \{0,1\}$ uniformly at random, the probability that f is computable by a circuit of size s is $2^{O(s\cdot\log(s))-2^n}$.*

Proof. There are 2^{2^n} functions on n bits. To upper-bound the number of size-s circuits, we show an injective mapping from this set of circuits to $\{0,1\}^{O(s\cdot\log s)}$. Specifically, each size-s circuit can be uniquely represented as a list of gates, where for each of the s gates we specify its type (constantly many bits) as well as the indices of the gates that feed into it ($O(\log s)$ bits). Hence there are only $2^{O(s\cdot\log s)}$ circuits. ■

The simple counting-based lower bound is actually tight (implying that the way of representing circuits described above is optimal, up to constants). This is because any function can be computed by a circuit of size $O(2^n/n)$; that is, the naive upper-bound

of $O(2^n)$ can be improved to match the lower bound. Combining the two results, the complexity of a random function is $\Theta(2^n/n)$.[6]

Theorem 9. *Every function $f\colon \{0,1\}^n \to \{0,1\}$ can be computed by a circuit of size $O(2^n/n)$.*

Proof Sketch. Consider a decision tree of depth n and size 2^n that computes f in the brute-force way, where all nodes at layer i read x_i.

The key observation comes from looking at the bottom-most layers of this decision tree, and noticing that there is a lot of waste in these layers. Specifically, let's look at the ℓ bottom layers, where ℓ is tiny (think of, say, $\ell = \text{loglog}(n)$). Each node g at layer $n-\ell$ is the root of a decision tree whose layers read $x_{n-\ell}, ..., x_n$. The key observation is that copies of the same decision tree appear again and again in the bottom ℓ layers; that is, many of the nodes g at level $n-\ell$ are roots of the same decision tree. This is since there are 2^{2^ℓ} possible functions on ℓ bits, but the number of nodes g is $2^{n-\ell}$.

The way to avoid the waste is to put aside $L = 2^{2^\ell}$ decision trees $T_1, ..., T_L$, which compute all ℓ-bit functions on $x_{n-\ell}, ..., x_n$. And now we connect each node g at level $n-\ell$ to the appropriate decision tree T_i. Note that (crucially) we re-use computation: The same tree T_i may be the child of many nodes $g, g', ...$ at level $n-\ell$.[7]

Balancing the parameters to avoid waste calls for $2^{2^\ell} \approx 2^{n-\ell}$, which we get using $\ell = \log(n - \log n)$. This yields a construction of size $O(2^{n-\ell}) = O(2^n/n)$. ∎

To road-test your familiarity with the definitions and the basic results, solve the following exercises at home.

Exercise 10. *Prove a size hierarchy theorem: For any $s(n) = o(2^n/n)$ there's a function computable in size s but not in size $o(s)$.*

Exercise 11. *Resolve the following "paradox": How could it be that there are 2^{2^n} functions, and still any function can be computed by a circuit of size $O(2^n/n)$? Isn't it information-theoretically impossible to compress an arbitrary 2^n-bit string to a smaller description, i.e. a circuit of size $O(2^n/n)$?*

Exercise 12. *Define a model of TMs that take "advice", i.e. for every input length n the machine has access to a best-case string a_n (where we don't bound the complexity of producing a_n). Define the class $\mathcal{DTIME}[T]/s$, where s bounds the advice length. Simulate $\mathcal{SIZE}[s]$ in $\mathcal{DTIME}[s']/s'$ and vice versa, with only a quasilinear simulation cost.*

References

[BF99] Harry Buhrman and Lance Fortnow. "One-Sided Versus Two-Sided Error in Probabilistic Computation". In: *Proc. 16th Symposium on Theoretical Aspects of Computer Science (STACS)*. 1999, pp. 100–109.

[6] We also know good bounds on the constant c inside the Θ, basically $1 - o(1) \leq c \leq 1 + o(1)$.

[7] Formally, this is not a decision tree anymore. Indeed, the only gap between this proof sketch and a full proof amounts to implementing the described "decision tree" as a circuit.

[BFN+93] László Babai, Lance Fortnow, Noam Nisan, and Avi Wigderson. "BPP has subexponential time simulations unless EXPTIME has publishable proofs". In: *Computational Complexity* 3.4 (1993), pp. 307–318.

[KL82] Richard M. Karp and Richard J. Lipton. "Turing machines that take advice". In: *L'Enseignement Mathématique. Revue Internationale. 2e Série* 28.3-4 (1982), pp. 191–209.

[Lau83] Clemens Lautemann. "BPP and the polynomial hierarchy". In: *Information Processing Letters* 17.4 (1983), pp. 215–217.

[Sip83] Michael Sipser. "A complexity theoretic approach to randomness". In: *Proc. 15th Annual ACM Symposium on Theory of Computing (STOC)*. 1983, pp. 330–335.

[Uhl74] Dietmar Uhlig. "On the synthesis of self-correcting schemes from functional elements with a small number of reliable elements". In: *Mathematical Notes of the Academy of Sciences of the USSR* 15 (1974), 558–562.

[Uhl92] Dietmar Uhlig. "Networks computing Boolean functions for multiple input values". In: *Boolean function complexity (Durham, 1990)*. Vol. 169. London Math. Soc. Lecture Note Ser. Cambridge Univ. Press, Cambridge, 1992, pp. 165–173.

[Wil10] Ryan Williams. "Improving exhaustive search implies superpolynomial lower bounds". In: *Proc. 42nd Annual ACM Symposium on Theory of Computing (STOC)*. 2010, pp. 231–240.