**CS 170: Efficient Algorithms and Intractable Problems** Spring 2020

Lecture Notes — April 23, 2020

*Profs. Alessandro Chiesa and Jelani Nelson*

## Lower Bounds

This class has mostly focused on, as the name of the class suggests, efficient algorithms for a wide variety of problems. For a given problem (whether it be maximum flow, the traveling salesman problem, or 3SAT), we identified computational resources of interest such as time or memory and developed algorithms that provided *upper bounds* on the amount of that resource required to solve the problem. More precisely, for a specific problem (say 3SAT) we can define a function

$$t(n) = \min_{\mathcal{A}} \max_{|x|=n} (\text{running time of } \mathcal{A} \text{ on input } x) \tag{1}$$

where the minimum is taken over all algorithms $\mathcal{A}$ that correctly solve the problem on every input, and the maximum is taken over all inputs of length $n$. Thus exhibiting a correct algorithm $\mathcal{A}$ for the problem and bounding $\mathcal{A}$'s worst case runtime, which is what we have typically done in this course, provides an *upper bound* on $t(n)$ (since the min is taken over all algorithms, and thus can only be lower than what is achieved by one particular algorithm).

The definition of $t(n)$ given above is what is called *non-uniform*: we are allowed to pick a different minimizer algorithm $\mathcal{A}$ for each $n$ in the definition of $t(n)$. We point out right now that the definition given above is actually somewhat problematic, since it would allow the description of $\mathcal{A}$ itself to grow with $n$. For example, the minimizer algorithm $\mathcal{A}_n$ for input size $n$ could simply be a lookup table that has the answer to all length-$n$ inputs in its source code. There are two main ways to get around this issue: (1) stick with a non-uniform version of the question, but in which for each $n$ you only minimize over $\mathcal{A}$ of bounded description complexity (e.g. bounding the length of its source code, or some other notion of complexity of $\mathcal{A}$), or (2) ask *uniform* questions. In uniform computation, essentially $\mathcal{A}$ is not allowed to change with $n$. A uniform question could for example be the following: for a specific function $f(n)$, say $f(n) = n^2$, is there one *fixed* algorithm $\mathcal{A}$ (to be used for all input sizes $n$) solving the problem correctly such that the worst-case running time of $\mathcal{A}$ on inputs of size $n$ is $O(f(n))$? Or does *every* fixed $\mathcal{A}$ solving the problem correctly have worst-case running time $\omega(f(n))$? For more in-depth treatment of uniform vs. non-uniform computation, take a course on complexity theory like CS 172 or CS 278. We will not dwell on these points further in this lecture; for the remainder of these notes we will focus on specific well-defined questions.

In order to ask the question of what the most efficient algorithm for a problem is, or to prove that every algorithm for a particular problem has running time $\omega(f(n))$ for some function $f$, we must first rigorously define what exactly an *algorithm* is. For this we need to have a *computational model* that describes how exactly computation works. We will not provide that rigorous definition in this course, but perhaps just raising this point will motivate you take take CS 172 (where you will learn about various such models, e.g. Turing machines, or deterministic finite state automata). We will though say that for a fairly general model of computation which allows implementation of all the algorithms covered in this class in roughly the running times we have proven, the *two-tape*

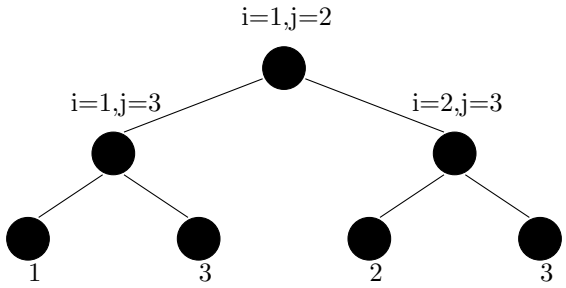


Figure 1: A comparison-based decision tree for computing the minimum element of a list of 3 elements. The root compares $x_1 \overset{?}{<} x_2$. If so, it follows the left child to compare $x_1 \overset{?}{<} x_3$, and otherwise it follows the right child to compare $x_2 \overset{?}{<} x_3$. The leaves are labeled with the index of the element that has the minimum value. The depth here is 2.

*Turing machine model*, humanity has so far been fairly awful in its ability to prove lower bounds. For example, whereas many people believe NP-complete problems like 3SAT and Knapsack require exponential time to solve, we have no lower bounds proven that preclude say an algorithm for 3SAT that solves the problem in $O(|\varphi|)$ time (where $\varphi$ is input formula and $|\cdot|$ denotes its description length), i.e. linear time! We *do* have problems for which we have superlinear lower bounds, but not for any problems covered in this class! For example, in CS 172 you will learn about the Time Hierarchy theorem, which for example implies that the following problem cannot be solved in linear time: given the source code $P$ for a program and an input $x$, does $P$ terminate on $x$ within $|x|^2$ steps? The trivial solution would be to simulate running $P$ on $x$ for at most $|x|^2$ steps, but of course that simulation would take at least quadratic time. It turns out it is possible to prove that not much better than such straightforward simulation is possible.

Though we do not get into the details of Turing machines in CS 170, we do below below survey a few other computational models and lower bounds in them. For the last model, communication complexity, we will actually show how some lower bounds are proven!

# 1 Comparison-based model

We discussed this model earlier in the semester when discussing sorting algorithms. An algorithm in this model operates on $n$ inputs that are comparable, $x_1, \ldots, x_n$, via some comparator operation we simply denote as "$<$". For simplicity we assume all these values are distinct, so $x_i \neq x_j$ for $i \neq j$. An algorithm is then modeled as a binary tree (a "decision tree") in which each node is labeled with some pair $i \neq j$. The algorithm's starting state corresponds to the root node. When in a state, the algorithm compares the two elements listed at that node, then follows the edge to the left child if $x_i < x_j$ and the right child otherwise (in which case $x_j < x_i$). The leaves of the tree then correspond to return values of the algorithm, i.e. when the algorithm reaches a leaf it knows the answer. The complexity of an algorithm, which is the worst-case number of comparisons on any input, is then the maximum depth of any leaf. See for example Figure 1.

As mentioned earlier in the semester, it is possible to prove that any comparison-based sorting algorithm takes $\Omega(n \log n)$ time, i.e. the depth of the corresponding decision tree is $\Omega(n \log n)$. This

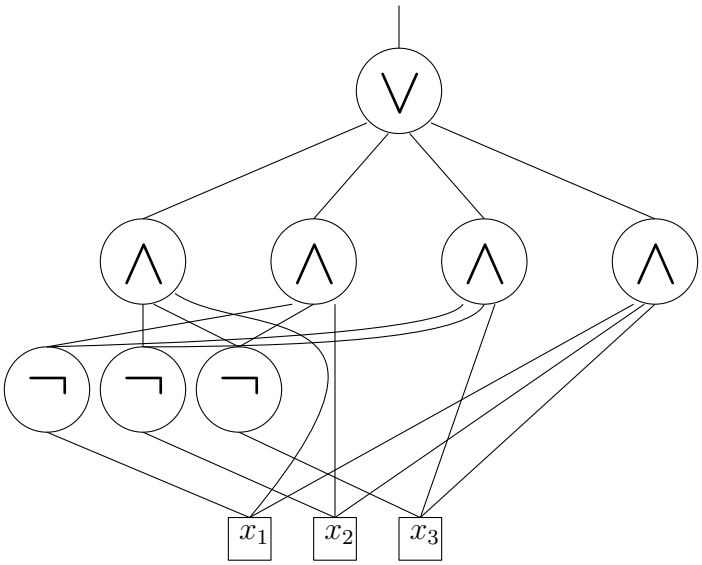


Figure 2: A circuit computing parity on 3 bits, by taking the OR of $(x_1 \wedge \bar{x}_2 \wedge \bar{x}_3)$, $(\bar{x}_1 \wedge x_2 \wedge \bar{x}_3)$, $(\bar{x}_1 \wedge \bar{x}_2 \wedge x_3)$, and $(x_1 \wedge x_2 \wedge x_3)$. Note $\neg x$ denotes $\bar{x}$, the negation of $x$.

is because the number of leaves of a depth-$d$ tree is at most $2^d$. Meanwhile there are $n!$ possibilities for what the true sorted order of the $n$ input elements is, and a correct sorting algorithm needs to tell which one is correct, i.e. it needs to have *at least* one distinct leaf in its decision tree for each sorted order. Thus we need $2^d \geq n!$, or $d \geq \log(n!)$. Since $n! \geq (n/2)^{n/2}$, this implies $d \geq (n/2)\log(n/2) = \Omega(n \log n)$ (note a better constant factor in the lower bound could be achieved by using Stirling's approximation to approximate $n!$).

We note here that this lower bound is against non-uniform computation as mentioned above, since it holds even though the "algorithm" (i.e. the decision tree) is allowed to change with $n$.

## 2 Circuit complexity

In this model of computation there are $n$ input bits that then feed as inputs into gates (AND, OR, and NOT) via wires. The gates then output the logical function they perform, which can be fed via a wire into other gates. The circuit should be acyclic, so that the output of a gate should not have a path back to its own input. There is then a final output gate, which has one output wire that gives the (Boolean) answer. See for example Figure 2, which gives a circuit for computing the parity of 3 bits, i.e. $x_1 \oplus x_2 \oplus x_3$. There are various notions of complexity one may wish to minimize for a circuit. The most popular two are *depth* (the length of the longest path from any $x_i$ to the output gate) and *size* (the total number of wires in the circuit). It is known that the number of circuits of size $s$ is $2^{O(s \log s)}$, whereas the number of distinct functions on $n$ bits is $2^{2^n}$ (there are $2^n$ possible inputs of size $n$, and each could be mapped to either 0 or 1). Thus to be able to capture all functions with size-$s$ circuits, we would need $2^{Cs \log s} \geq 2^{2^n}$, which by taking logarithms and rearranging implies there do exist functions that require circuit size exponential in $n$, specifically of size $s = \Omega(2^n/n)$. Despite knowing that such a function *exists*, humanity still has

not yet managed to come up with an *explicit* function that requires exponential size circuits. In fact, the best known lower bound for any explicit function is $s \geq (3 + 1/86)n$ [1]. One reason that computer scientists are interested in circuit complexity is the following: let P/poly denote the class of all problems that can be solved by circuits of size $s(n) \leq poly(n)$ for size-$n$ inputs. Then it is known that P$\subset$P/poly. Thus if we can show that some NP-complete problem is *not* in P/poly, such as say 3SAT, we will have shown that P$\neq$NP! Of course we are quite far from doing this, since to date we have not been able to show an even superlinear lower bound for any explicit problem, let alone a super-polynomial one.

# 3 Cell probe model

This model is primarily used to model data structures (and specifically to prove lower bounds against update time, query time, and/or memory). The data structure has a *memory* `M` consuming $S$ machine words of space. Each word is $w$ bits (think for example $w = 32$ or 64 on modern architectures). There is then the query/update algorithm of the data structure. Whenever the data structure receives an operation request, it must do some computation in addition to memory read/writes to process it. In the cell probe model though, *only memory read/writes are counted*. That is, to process an operation there will be several rounds of communication between the algorithm and the memory. Each "message" from the algorithm to to the memory will either be `read`$(i)$, in which case the memory responds with `M`$[i]$, or it will be `write`$(i, \Delta)$, in which case the memory will perform the change `M`$[i] \leftarrow \Delta$. Here $i$ is $\lceil \log_2 S \rceil$ bits, and $\Delta$ is $w$ bits. Some computation may have been performed to decide which cells to read/write, but they are not counted. The number of rounds of communication required to perform an operation is precisely the number of memory read/writes of the data structure. Thus we denote the number of back-and-forth communciation rounds as $t_u$ for updates (the "update time") and $t_q$ for queries (the "query time").

The cell probe model should be compared with the *Word RAM* model, which is similar but also takes into account computation. That is, not only do read/write each take 1 unit of time in the Word RAM model, but so do other architecture-supported operations on single words (e.g. `+`, `-`, `*`, `/`, $\ll$, $\gg$, $|$, `load` (i.e. read), `store` (i.e. write), etc.). These operations are *free* in the cell probe model, since the only operations that count are `load` and `store` (i.e. read and write) and all other computation that led to the decision of what to read/write is free! Thus the best complexity to solve a data structural problem in the cell probe model is a *lower bound* on the complexity to solve it in the Word RAM model (which is the standard model usually studied for algorithms in CS 170), since not having to count computation other than `load`/`store` can only make the data structure faster!

As already evident from the language used, the cell probe model is essentially a two-player "communication game". One player is the algorithm, whose input is the description of an operation (update or query). The other player is the memory, whose input are the memory contents. They communicate back and forth in rounds, where the algorithm's messages are at most $\lceil \log_2 S \rceil + w$ bits long, and the memory's messages are at most $w$ bits long. For specific data structural problems, one then wants to understand the minimum number of rounds of communication required to solve the problem, i.e. support some sequence of operations. Tools from communication complexity naturally turn out to be relevant to prove lower bounds on the number of rounds, i.e. time, needed; communication complexity will be further discussed in Section 5. Lower bounds for several data structural problems have been proven via cell probe lower bounds, such as union-find, predecessor

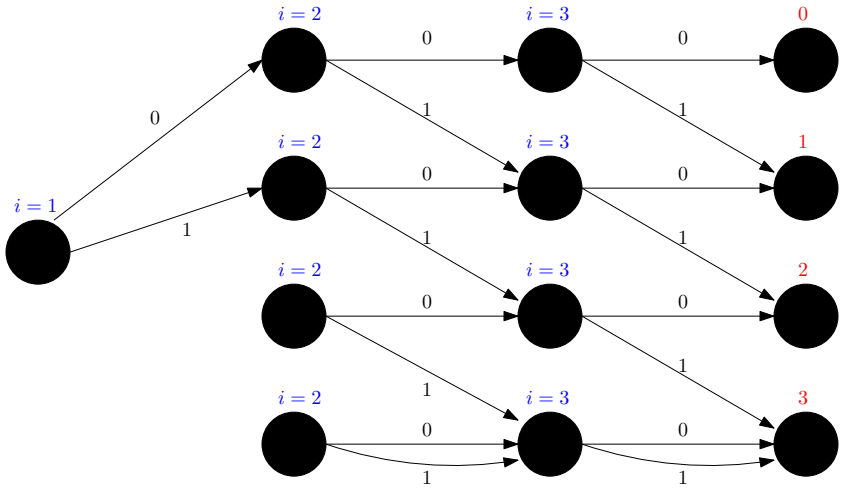


Figure 3: A layered branching program with $L = 3$ and width 4 which computes $f(x_1, x_2, x_3) = x_1 + x_2 + x_3$. The sinks in the final layer are labeled with their answer in red (here $Y = \{0, 1, 2, 3\}$ is the set of possible sums of 3 bits). Each non-sink node is labeled in blue with the input $x_i$ it reads; in this example each node in layer $i$ reads $x_{i+1}$, but in general nodes in the same layer are not required to read the same bit, and also in general $L$ may be bigger than $n$ and we may read the same bit multiple times in a path. The above branching program captures the simple algorithm which just sums the bits in order and keeps a running sum; the nodes in the $j$th row from the top, $j = 0, 1, 2, 3$, represent that the current running sum is $j$.

search, dynamic partial sums, and many more. In fact, cell probe lower bounds are the golden standard for proving data structure lower bounds, as they do not depend on exactly which instructions are supported by the machine architecture, as only reads and writes are counted.

# 4 Branching programs

A branching program is a model of computation for computing some function $f : \{0, 1\}^n \to Y$. A branching program is a directed acyclic graph with a unique source node and some number of sink nodes. Each sink node is labeled with one element of $Y$ (the return value for the inputs that lead there). Each non-sink node is further labeled with an index $i \in \{1, \ldots, n\}$ and has two outgoing edges: one labeled 0 and one labeled 1. Any input $x \in \{0, 1\}^n$ then defines a path through this graph: starting at the sink, we iteratively read $x_i$ for the $i$ the current node is labeled with, then we follow the edge out of that node labeled with $x_i$. We keep walking through the graph in this way until we arrive at a sink, at which point we return the answer that the sink is labeled with. We let $L$ denote the *length* of a branching program, which is the maximum number of edges in a path from the source to some sink.

A sub-case of branching programs of particular interest is that of *layered branching programs*. In a layered branching program, one can imagine that the vertices are partitioned into "layers" $0, \ldots, L$. The source is in layer 0 and the sinks are in layer $L$, and nodes in layers $0 \leq i < L$ only have edges to nodes in layer $i + 1$. The *width* $W$ of a layered branching program is the maximum number of nodes in any layer. Since for any fixed layer the "program" can be in one of $W$ states, the state

of the program can be represented in $\lceil \log_2 W \rceil$ bits, which is thus a proxy for the "space" (in bits) used by the algorithm. The number of layers $L$ is then a proxy for "worst-case running time". Figure 3 gives an example of a layered branching program with $L = 3, W = 4$ to compute the sum of 3 input bits. Proving length/width tradeoffs for branching programs computing a specific function thus corresponds to time/space tradeoffs for computation.

# 5 Communication complexity

The final model we talk about today in which one can prove lower bounds is *communication complexity*. In 2-player communication complexity there is a player Alice holding some input $x \in \mathcal{X}$ and another player Bob holding $y \in \mathcal{Y}$. They both know ahead of time a function $f : \mathcal{X} \times \mathcal{Y} \to \{0,1\}$ and would like to compute $f(x,y)$. This works by Alice first sending a message to Bob (based on her input $x$), who then responds by sending a message back to Alice (based on his input $y$ as well as the message he just received from Alice), etc. back and forth. The *complexity* of a communication protocol to compute $f(x,y)$ is then the total communication in bits, i.e. the sum of all message lengths before the answer is computed. For a function $f$, we let $D(f)$ denote the worst-case cost of an optimal (i.e. least number of bits communicated) deterministic protocol that allows Alice and Bob to compute $f(x,y)$ for every pairs of inputs $x, y$.

Communication complexity lower bounds have applications across many areas of computer science. For example, Karchmer and Wigderson in 1990 showed that the minimum depth of a circuit computing a function $f : \{0,1\}^n \to \{0,1\}$ is exactly equal to the optimal communication complexity of a certain communication problem (a "Karchmer-Wigderson game") they defined involving $f$: namely Alice gets an $x$ such that $f(x) = 0$, Bob gets a $y$ such that $f(y) = 0$, and the players must communicate to find an $i$ such that $x_i \neq y_i$. As seen above when discussing the cell probe model, communication complexity lower bounds can also yield data structure lower bounds. Also, communication lower bounds can provide lower bounds for streaming algorithms as discussed in last lecture, as we will now show.

Specifically, we will focus on the distinct elements problem from last lecture: there is a stream $x_1, \ldots, x_m$ of integers in $\{1, 2, \ldots, n\}$, and we must compute the number of *distinct* integers in the stream. We will show that communication complexity can be used to provide lower bounds on the optimal memory complexity of solving this problem.

**Fact 1.** *Let* $\mathsf{EQ}_n : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ *denote the equality function on* $n$ *bit strings, i.e.* $\mathsf{EQ}_n(x,y) = 1$ *iff* $x = y$. *Then* $D(\mathsf{EQ}_n) = n$.

We now show that Fact 1 implies lower bounds on the memory required to solve distinct elements.

**Lemma 2.** *Any correct deterministic algorithm for the distinct elements problem uses at least* $n$ *bits of memory.*

*Proof.* Let $\mathcal{A}$ be a streaming algorithm for exact computation of the number $t$ of distinct elements, using space $S$. We will show that the existence of such an algorithm implies a communication protocol for $\mathsf{EQ}_n$ with $S$ bits of communication. Since any protocol requires at least $n$ bits (Fact 1), this gives the inequality $S \geq n$ as desired.

The protocol is as follows: Alice feeds all the $i$ such that $x_i = 1$ into $\mathcal{A}$ as a stream. She then sends the memory contents of $\mathcal{A}$ ($S$ bits) as a message to Bob. Bob then queries $\mathcal{A}$ for the number $r$ of distinct elements so far, which is the number of 1 bits in $x$. If $r$ doesn't match the number of 1 bits in $y$, Bob can report that $x \neq y$. Otherwise, he continues running $\mathcal{A}$ from the state in which it left off (which he can do since Alice sent its memory contents) and feeds it all $i$ such that $y_i = 1$ as a stream. Bob then queries the algorithm again and declares $x = y$ iff it again says $r$. This is correct since if $x \neq y$, since $x, y$ have the same number of 1 bits, $y_j$ must be 1 for some $j$ with $x_j = 0$. Thus if $x \neq y$, the number of distinct elements would increase to become larger than $r$. □

The above lower bound was for deterministic and exact computation. What if we are allowed to have approximation though? Specifically, if $t$ is the true number of distinct elements, what if our algorithm promises to return a value $\tilde{t}$ guaranteed to satisfy $t \leq \tilde{t} \leq C \cdot t$, where $C$ is some approximation factor?

**Fact 3.** *For every $n \geq 1$ there exists a collection $\mathcal{B}_n$ of subsets of $\{1, \ldots, n\}$ such that (1) $|\mathcal{B}_n| \geq 2^{cn}$ for some universal constant $c$, (2) all $B \in \mathcal{B}_n$ have the same size $b > 0$, and (3) if $B \neq B' \in \mathcal{B}_n$, then $|B \cap B'| \leq |B|/10 = b/10$. Furthermore, if $\mathsf{EQ}_{\mathcal{B}_n}$ is the equality problem but when promised that $x, y$ are indicator vectors of elements of $\mathcal{B}_n$, then $D(\mathsf{EQ}_{\mathcal{B}_n}) = \Omega(n)$.*

**Lemma 4.** *Any deterministic streaming algorithm for $C$-approximate distinct elements computation for $C < 1.1$ requires $\Omega(n)$ bits of memory.*

*Proof.* We show that such an algorithm $\mathcal{A}$ using space $S$ implies the existence of a protocol for $\mathsf{EQ}_{\mathcal{B}_n}$ with $S$ bits total total communication, and thus $S \geq D(\mathsf{EQ}_{\mathcal{B}_n}) = \Omega(n)$.

The protocol is similar with the proof of Lemma 2. Alice runs $\mathcal{A}$ on all the $i$ such that $x_i = 1$ then sends its memory contents as a message to Bob. Bob then continues to feed all the $i$ such that $y_i = 1$ to $\mathcal{A}$. He then queries the approximate number of distinct elements $\tilde{t}$ and declares $x = y$ iff $\tilde{t} \geq 1.9b$. This is correct since if $x = y$ then the true number $t$ of distinct elements is $b$ and thus $b \leq \tilde{t} \leq 1.1b$. Meanwhile if $x \neq y$ then let $B = \{i : x_i = 1\}$ and $B' = \{i : y_i = 1\}$ so that $B, B' \in \mathcal{B}_n$. Then the true number of distinct elements is $|B \cup B'| = |B| + |B'| - |B \cap B'| = 2b - |B \cap B'| \geq 1.9b$. Thus $1.9b \leq \tilde{t} \leq 1.1 \cdot 1.9b$. □

It is also possible to use communication complexity to prove that any *exact* randomized Monte Carlo streaming algorithm for distinct elements also needs $\Omega(n)$ bits of space. Thus any low-memory algorithm needs to be both randomized *and* approximate, such as the algorithm discussed in the previous lecture.

## References


[1] Alexander Golovnev, Edward A. Hirsch, Alexander S. Kulikov. A Better-Than-3n Lower Bound for the Circuit Complexity of an Explicit Function. *Proceedings of the 57th Annual IEEE Symposium on the Foundations of Computer Science (FOCS)*, pages 89–98, 2016.

[2] Mauricio Karchmer, Avi Wigderson. Monotone Circuits for Connectivity Require Super-Logarithmic Depth. *SIAM J. Discrete Math.* 3(2), pages 255–265, 1990.