CS1200: Intro. to Algorithms and their Limitations Anshu & Vadhan

Lecture 18: Computational Complexity

Harvard SEAS - Fall 2024 *2024-11-05*

1 Announcements

- Recommended Reading: MacCormick §5.3–5.5, Ch. 10, 11
- Reminder: ps7 is *not* due tomorrow.
- Late day tracking & penalities: see email and Ed post from Maxwell.
- ps5 revision videos due today
- ps6 feedback:
  - response rates dropping!
  - median time among respondents: 7hrs college, 8hrs DCE
  - enjoyed Lyber problem, can use more clarity on Q1
  - Lecture pacing just right (76%), too fast (24%)
  - enjoy real-life connection
  - positive feedback on section and OH

2 Loose Ends from Lec17

2.1 The Extended (or Strong) Church–Turing Thesis

The Church–Turing thesis only concerns problems solvable at all by these models of computation (Word-RAM programs, etc.). We haven't even seen any problems that are *not* solvable by Word-RAM programs—that will be a topic for the end of the course. There is, however, a stronger version of the Church–Turing hypothesis that also covers the efficiency with which we can solve problems.

> Extended Church–Turing Thesis v1: Every physically realizable model of computation can be simulated by a Word-RAM program with only a *polynomial* slowdown in runtime. Conversely, Word-RAM programs can be physically simulated in real time only polynomially slower than their defined runtime.

The Strong Church–Turing Thesis is not a precise mathematical claim, and thus cannot be formally proven. In fact, randomized algorithms, massively parallel computers, and quantum computers all could potentially provide an exponential savings in runtime. (For randomized algorithms, however, it is conjectured that they provide only a polynomial savings, as discussed in Lecture 8.)

If we modify the statement with some qualifiers, then these challenges no longer apply:

> Extended Church–Turing Thesis v2: Every physically realizable, deterministic, and sequential model of computation can be simulated by a Word-RAM program with only a polynomial slowdown in runtime. Conversely, Word-RAM programs can be physically simulated in a deterministic and sequential manner in real time only polynomially slower than their defined runtime.

"Deterministic" rules out both randomized and quantum computation, as both are inherently probabilistic. "Sequential" rules out parallel computation. This form of the Extended Church–Turing Thesis has stood the test of time for the approximately fifty years since it was formulated, even as computing technology has changed tremendously in that time.

Note: in contrast to the above claims about Word-RAM, we had a pset where a natural attempt at simulating RAM programs on real computers incurred an exponential slowdown, and our RAM to Word-RAM simulation theorem also has a slowdown factor that can get exponentially large (due to bitlength of numbers computed). So, the choice of base model (Word-RAM) is important here in a way it isn't for the regular Church–Turing Thesis.

2.2 Turing Machines (informal)

The Extended Church–Turing Thesis is usually stated in terms of *Turing Machines*, rather than Word-RAM programs. Turing Machines were the model of computation introduced in Turing's landmark 1936 paper that is considered one of the founding papers of the field of computer science. At the time, there were no general-purpose physical computers, and Turing's goal was to capture the intuitive notion of "computability." Turing's model and his results about it (especially the Universal Turing Machine, which we will see in a couple of weeks) were a major inspiration for the development of physical computers. Turing Machines can be seen as a variant of the (Word-)RAM model with *constant* word size. Formal definitions are given below in the optional reading, but the main changes are as follows:

1. *Finite Alphabet:* Each memory cell and variable can only store an element from $[q]$ for a finite *alphabet size* q, which is independent of the input length and does not grow with the computation's memory usage. (Elements of $[q]$ can be stored using words of length $w = \lceil \log_2 q \rceil = O(1)$.)

2. *Memory Pointer:* In addition to the variables, there is a separate `mem_ptr` that stores a natural number, pointing to a memory location, initialized to $\texttt{mem\_ptr} = 0$.

3. *Read/write:* Reading and writing from memory is done with commands of the form $\texttt{var}_i = M[\texttt{mem\_ptr}]$ and $M[\texttt{mem\_ptr}] = \texttt{var}_i$, instead of using $M[\texttt{var}_j]$.

4. *Moving Pointer:* There are commands $\texttt{mem\_ptr} = \texttt{mem\_ptr} + 1$ and $\texttt{mem\_ptr} = \texttt{mem\_ptr} - 1$ to increment and decrement `mem_ptr`.

With these changes, there is a mathematically very elegant description of Turing Machines (see optional reading in the detailed lecture notes and the recommended textbooks), with no arbitrary set of operations being chosen (allowing any "constant-sized" computation to happen in one step). For this reason, Turing machines are the main model covered in classes like CS1210 on the Theory of Computation. We work with the Word-RAM Model because it is better-suited for measuring the efficiency of algorithms in practice, and is the main model used (implicitly) in courses on algorithms.

On the surface, Turing Machines seem to be weaker than Word-RAM programs. Nevertheless they are polynomially equivalent:

Theorem 2.1. *Turing Machines and Word-RAM Programs can simulate each other with only a polynomial slowdown.*

A proof sketch can be found in the optional reading. Because of the above, the Word-RAM Model and Turing Machines are referred to *strongly Turing-equivalent* models of computation.

2.3 Turing Machine Formalism (optional)

This material was not covered in lecture but is optional reading in case you are interested.

We work our way to Turing Machines by first modifying the (Word-)RAM model to have an $O(1)$ word size (independent of the input length or memory usage):

Definition 2.2 (TM-RAM programs)**.** A *TM-RAM* program P is like a RAM program with the following modifications:

1. *Finite Alphabet:* Each memory cell and variable can only store an element from $[q]$ for a finite *alphabet size* q, which is independent of the input length and does not grow with the computation's memory usage.
2. *Memory Pointer:* In addition to the variables, there is a separate `mem_ptr` that stores a natural number, pointing to a memory location, initialized to $\texttt{mem\_ptr} = 0$.
3. *Read/write:* Reading and writing from memory is done with commands of the form $\texttt{var}_i = M[\texttt{mem\_ptr}]$ and $M[\texttt{mem\_ptr}] = \texttt{var}_i$, instead of using $M[\texttt{var}_j]$.
4. *Moving Pointer:* There are commands $\texttt{mem\_ptr} = \texttt{mem\_ptr} + 1$ and $\texttt{mem\_ptr} = \texttt{mem\_ptr} - 1$ to increment and decrement `mem_ptr`.

See Figure 1.

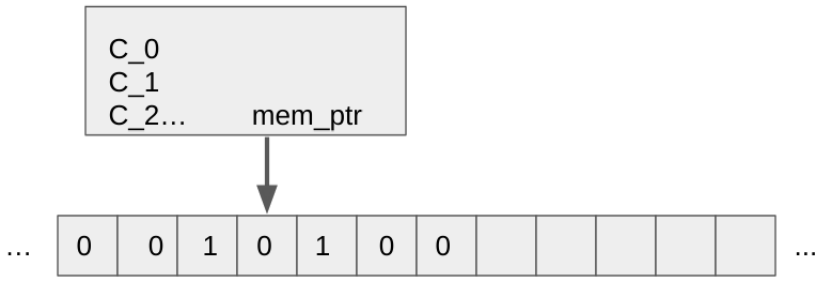

Figure 1: A TM RAM machine, with memory pointer and commands.

Philosophically, TM-RAM programs are appealing because one step of computation only operates on constant-sized objects (ones with domain $[q]$). However, as we will discuss below, the ability to only increment and decrement `mem_ptr` by 1 does make TM-RAM programs somewhat slow compared to Word-RAM programs.

We can think of a TM-RAM as being described by its infinite memory array, which is often called a *tape*, together with a *finite-state machine* that keeps track of the values of all the variables and the current line number of the program. This finite-state machine moves right and left along the tape as `mem_ptr` is incremented and decremented. Note that the number of possibilities for the state of a TM-RAM's computation, excluding the memory contents is $q^k \cdot \ell$, if there are k variables and ℓ lines in the program, which is indeed a constant independent of the size of the input. In each time step, the TM-RAM may read from or write to its current location on the tape and update its state (by any changes to values of variables and the current line number).

Thus, we can *simulate* a TM-RAM by a *Turing Machine* described as follows:

Definition 2.3 (Turing machine)**.** A *Turing machine* $M = (Q, \Sigma, \delta, q_0, H)$ is specified by:

1. A finite set Q of states.
2. A finite alphabet Σ (e.g. $[q]$).
3. A transition function $\delta : Q \times \Sigma \to Q \times \Sigma \times \{L, R, S\}$.
4. An initial state $q_0 \in Q$.
5. A set $H \subseteq Q$ of halting states.

The semantics of the transition function δ are as follows: $\delta(q, \sigma) = (q', \sigma', m)$ means that if the current state is q and the current location on the tape has value σ, then we transition to state q', overwrite the current tape location with σ' and move along the tape according to whether $m = R$ ("move right"), $m = L$ ("move left"), $m = S$ ("stay in place"). You can find a precise definition of how Turing Machines compute (in particular, how their input and output is specified) in the recommended textbooks (e.g. MacCormick or Sipser).

Conversely, it is also possible to show that TM-RAMs can simulate TMs. Thus we have:

Theorem 2.4 (Equivalence of TMs and TM-RAMs)**.**

1. *For every TM-RAM program P, there is a Turing Machine M such that for all inputs x, M halts on x iff P halts on input x, and if they do halt on x, then $M(x) = P(x)$ and $T_M(x) = O(T_P(x))$.*
2. *For every Turing Machine M, there is a TM-RAM program P such that for all inputs x, P halts on x iff M halts on input x, and if they do halt on x, then $P(x) = M(x)$ and $T_P(x) = O(T_M(x))$.*

Thus Turing Machines are indeed equivalent to a restricted form of RAM programs. The appeal of Turing machines is their mathematically simple description, with no arbitrary set of operations being chosen (allowing any "constant-sized" computation to happen in one step).

What about Turing Machines vs. Word-RAM Programs? It's intuitive and not too difficult to simulate TM-RAM programs (and hence TMs) by Word-RAM programs; the larger word size of Word-RAM only helps. Simulating Word-RAMs by TMs takes more work and incurs a substantial slowdown.

Theorem 2.5. *For every Word-RAM Program P, there is a TM-RAM program P' such that for all inputs x, P' halts on x iff there is a word length w such that $P[w](x)$ halts on x without crashing,*

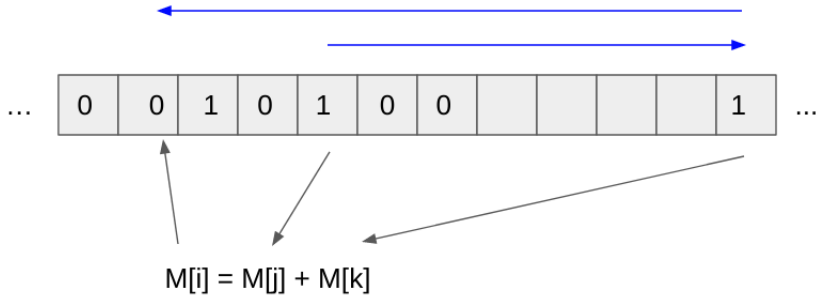

Figure 2: The requirement to move the memory pointer step by step in TM-RAM induces an up to quadratic slowdown vs RAM.

and if they do halt, $P'(x) = P[w](x)$ *and*

$$T_{P'}(x) = O\left(T_P(x) \cdot S_P(x) \cdot w_P(x)^3\right),$$

where $S_P(x)$ *is the number of MALLOCed memory locations used by* P *on input* x*, and* $w_P(x)$ *is the minimum non-crashing word length for* P *on* x*.*

Proof Sketch.
Our TM-RAM will try different word lengths $w = 1, 2, \ldots$ for P until it finds one where $P[w](x)$ does not crash. For each word length w, it simulates P as follows:

- Memory of P: encoded as $S \cdot w$ bits in the memory of P', at a point in the computation when P uses S memory cells and has a word size of w.
- Values of the variables of P: These take $O(w)$ bits and are stored in memory locations of P' near `mem_ptr`, and copied (using $O(w^2)$ steps) every time P' wants to move `mem_ptr` to a different simulated memory location of P.
- Simulating one step of P: P' scans over its entire $S \cdot w$-bit memory, doing updates (arithmetic operations, read/write operations, possibly increasing S, etc.) and copying the values of its variables as it goes. This takes time $O(S \cdot w^2)$.

Thus if P runs for T steps, the entire simulation for a fixed value of w takes time

$$O(T \cdot (S \cdot w^2)).$$

Trying $w = 1, 2, \ldots$ costs an extra factor of $w_P(x)$ in the runtime.

□

We can bound the slowdown of the above simulation solely in terms of $T_P(x)$ and the size of the input x (both its length n as an array and the magnitude of the numbers $x[i]$ in the array). Recall

that $S_P(x) \leq n + T_P(x)$, since a Word-RAM program can call MALLOC at most $T_P(x)$ times. In addition, we have:

$$\begin{aligned} w_P(x) &\leq O\left(\log\left(\max\left\{S_P(x), x[0], \ldots, x[n-1]\right\}\right)\right) \\ &= O\left(\log \max\left\{n + T_P(x), x[0], \ldots, x[n-1]\right\}\right). \end{aligned}$$

Thus, in the standard case that $T_P(x) \geq n$, so that P has enough time to read its input, we have

$$T_{P'}(x) = O\left(T_P(x)^2 \cdot \log^3\left(\max\{T_P(x), x[0], \ldots, x[n-1]\}\right)\right)$$

which is nearly quadratic in $T_P(x)$ with an addition blow-up that is cubic in the bit-length of the elements $x[i]$ of the input. (Note the contrast with the RAM to Word-RAM simulation, where the slowdown was polynomial in the bit-length of the numbers constructed during the computation of the RAM program, which could be exponentially bigger than those in input.)

So TM-RAMs and Turing Machines can simulate Word-RAM programs, but with only a polynomial slowdown in runtime. This is a lot better than the relation between RAM programs and Word-RAM programs, which incurs an exponential slowdown in simulating the former by the latter.

3 Computational Complexity

Computational complexity aims to classify problems according to the amount of resources (e.g. time) that they require to solve.

For example, we've seen problems whose fastest *known* algorithms run in:

- Linear time: ShortestPaths, 2-Coloring in time $O(n + m)$.
- Nearly linear time: Sorting, Interval Scheduling (Decision, Optimization, Coloring) in time $O(n \log n)$.
- Polynomial time: Bipartite Matching in time $O(nm)$, 2-SAT in time $O(n^3)$.[1]
- Exponential time: k-Coloring for $k \geq 3$, k-SAT for $k \geq 3$, Independent Set, and Longest Path in time $O(c^n)$ for constants $c > 1$.

Recall that the same problem can have many different algorithms each with a different runtime. For example, for SortingOnAFiniteUniverse, we have seen algorithms whose runtimes are $O(n \log n)$ (MergeSort), $O(n+U)$ (SingletonBucketSort), $O(n+n \cdot (\log U)/(\log n))$ (RadixSort), and $O(n \cdot n!)$ (ExhaustiveSearchSort). An ultimate goal of Computational Complexity would be to identify the *fastest* possible runtime for a problem, so that once we have achieved it, we can stop trying to improve our algorithms further. As we will see, the field is not quite there yet, but we do have a very rich understanding of the complexity of problems.

To develop a robust and clean theory for classifying problems according to computational complexity, we make two choices:

- A problem-independent size measure. Recall that we allowed ourselves to use different size parameters for different problems (array length n and universe size U for sorting; number n of vertices and number m of edges for graphs, number n of variable and number m of clauses for

[1] A linear-time algorithm for 2-SAT is actually known, based on DFS (which is covered in CS 1240).

Satisfiability). To classify problems, it is convenient to simply measure the size of the input x by its *length N in bits*, which we call its *bitlength* (or sometimes just *length*) and denote by $|x|$. For example:

  - Array of n numbers from universe size U: $N = \Theta(n \log_2 U)$.
  - Graphs on n vertices and m edges in adjacency list notation: $N = \Theta((n+m)\log n)$.
  - 3-SAT formulas with n variables and m clauses: $N = \Theta(m \log n)$.

- Polynomial slackness in running time: We will only try to make coarse distinctions in running time, e.g. polynomial time vs. super-polynomial time. If the Extended Church-Turing Thesis is correct, the theory we develop will be independent of changes in computing technology. It is possible to make finer distinctions, like linear vs. nearly linear vs. quadratic, if we fix a model (like the Word-RAM), and a newer subfield called *Fine-Grained Complexity* does this.

To this end, we define the following *complexity classes*.

Definition 3.1 (complexity classes)**.** For a function $T : \mathbb{N} \to \mathbb{R}^+$,

- $\mathsf{TIME}_{\mathsf{search}}(T(N))$ is the class of computational problems $\Pi = (\mathcal{I}, \mathcal{O}, f)$ such that there is a Word-RAM program solving Π in time $O(T(N))$ on inputs of bitlength N.
- $\mathsf{TIME}(T(N))$ is the class of *decision* (i.e. yes/no) problems in $\mathsf{TIME}_{\mathsf{search}}(T(N))$.
- (Polynomial time)
$$\mathsf{P}_{\mathsf{search}} = \bigcup_{c \geq 0} \mathsf{TIME}_{\mathsf{search}}(n^c), \qquad \mathsf{P} = \bigcup_{c \geq 0} \mathsf{TIME}(n^c)$$
- (Exponential time)
$$\mathsf{EXP}_{\mathsf{search}} = \bigcup_{c \geq 0} \mathsf{TIME}_{\mathsf{search}}\left(2^{n^c}\right), \qquad \mathsf{EXP} = \bigcup_{c \geq 0} \mathsf{TIME}\left(2^{n^c}\right).$$

Note that $\mathsf{P}_{\mathsf{search}}$ and P would be the same if we replace Word-RAM with any strongly Turing-equivalent model, like Turing Machines. (Remark on terminology: what we call $\mathsf{P}_{\mathsf{search}}$ is called Poly in the MacCormick text, and is often called FP elsewhere in the literature.)

By this definition, ShortestPaths, 2-Coloring, Sorting, IntervalScheduling, BipartiteMatching, and 2-SAT are all in $\mathsf{P}_{\mathsf{search}}$ (as well as P for decision versions of the problems). However, all we know to say about 3-Coloring, 3-SAT, IndependentSet, or LongestPath is that they are in $\mathsf{EXP}_{\mathsf{search}}$. Can we prove that they are not in $\mathsf{P}_{\mathsf{search}}$?

The following seems to give some hope:

Theorem 3.2. $\mathsf{P}_{\mathsf{search}} \subsetneq \mathsf{EXP}_{\mathsf{search}}$*, and* $\mathsf{P} \subsetneq \mathsf{EXP}$.

Here $A \subsetneq B$ means that A is a *strict* subset of B. That is, every element of A is in B, and there is some element $b \in B \setminus A$. We won't give a proof of the strict inclusion (take CS 1210 for that), but we'll see similar proofs in the last unit of the course. We even know an example of a problem in $\mathsf{EXP}_{\mathsf{search}} \setminus \mathsf{P}_{\mathsf{search}}$ (in fact $\mathsf{EXP} \setminus \mathsf{P}$), which we will mention (without proof) in the last unit of the course. (It is the "Bounded Halting Problem.") Unfortunately, we do not know how to prove that many of the problems we care about (like 3-SAT, 3-Coloring, LongestPath, IndependentSet) are in $\mathsf{EXP}_{\mathsf{search}} \setminus \mathsf{P}_{\mathsf{search}}$. However, what we *can* do is relate the seeming hardness of problems to each other (and many others!) via *reductions*.

4 Polynomial-Time Reductions

Definition 4.1. For computational problems Π and Γ, we write $\Pi \leq_p \Gamma$ if there is a *polynomial-time* reduction R from Π to Γ. That is, there is a constant $c \geq 0$ such that R runs in time at most $O(N^c)$ on inputs of length N, counting oracle calls as one time step. Equivalently, there is a constant $d \geq 0$ such that $\Pi \leq_{O(N^d),O(N^d)\times O(N^d)} \Gamma$.

Q: Why does the equivalence stated at the end of the definition hold?

A: A reduction that runs in time at most $T_R(N) = O(N^c)$ can make at most $T_R(N) = O(N^c)$ oracle calls, and each of those oracle calls is an array whose length is at most the length of its MALLOC'ed memory, which is at most $n + T_R(N) = O(N^c)$, since $n \leq N$ and assuming that $c \geq 1$ without loss of generality. The bitlength of the oracle calls is thus at most $O(N^c) \cdot w$, where w is the minimum non-crashing word length. Recall that

$$w = O\left(\log\left(\max\left\{n + T_R(N), x[0], \ldots, x[n-1]\right\}\right)\right) = O\left(\log\left(\max\left\{O(N^c), 2^N - 1\right\}\right)\right) = O(N),$$

where we used $x[i] \leq 2^N - 1$ because x, and hence each of its entries, is at most N bits long. Thus, the bitlength of the oracle queries is at most $O(N^c) \cdot O(N) = O(N^{c+1})$,

Some examples of polynomial-time reduction that we've seen include:

- 3-Coloring $\leq_p$ SAT
- LongPath $\leq_p$ SAT