CS120: Intro. to Algorithms and their Limitations Hesterberg & Vadhan

Lecture 18: Computational Complexity

Harvard SEAS - Fall 2022 *2022-11-02*

1 Announcements

Recommended Reading:

- MacCormick §5.3–5.5, Ch. 10, 11
- Salil is away. Adam office hours today in 2.122

2 Computational Complexity

A common category error when discussing computational problems is to talk about the "runtime of the problem", when runtime is a property of an algorithm, not of a problem. For instance, sorting is a problem which we've seen solved by algorithms whose runtimes are $O(n \log n)$ (Merge Sort), $O(n+U)$ (Radix Sort), and $O(n!n)$ (brute force). (Note that in this case there isn't even a single best runtime!)

There is a sense in which we can talk about runtime (or space, or probability of correctness) of a problem: we say that a problem is solvable in time $O(T(n))$ if there exists an algorithm which solves it that quickly. Note that proving that a problem *is* solvable in time $O(T(n))$ is straightforward (give a single algorithm solving it in time $O(T(n))$), but saying that a problem *is not* solvable in time $O(T(n))$ requires knowing that *no* algorithm with runtime $O(T(n))$ solves it, a much harder claim.

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

For example, we've seen algorithms that are:

- Linear time: Shortest Paths, 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 $c > 1$.

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

[1] A linear-time algorithm for 2-SAT is actually known, based on DFS (which is covered in CS 124) rather than BFS/Reachability.

- 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 Satisfiability). To classify problems, it is convenient to simply measure the size of the input by its *length N in bits.* 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 2.1.

- 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 bit-length N.

  $\mathsf{TIME}(T(N))$ is the class of *decision* (i.e. yes/no) problems in $\mathsf{TIME}_{\mathsf{search}}(T(N))$.
- (Polynomial time)

$$\mathsf{P}_{\mathsf{search}} = \bigcup_c \mathsf{TIME}_{\mathsf{search}}(n^c), \qquad \mathsf{P} = \bigcup_c \mathsf{TIME}(n^c)$$

- (Exponential time)

$$\mathsf{EXP}_{\mathsf{search}} = \bigcup_c \mathsf{TIME}_{\mathsf{search}}\left(2^{n^c}\right), \qquad \mathsf{EXP} = \bigcup_c \mathsf{TIME}\left(2^{n^c}\right).$$

(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.)

Note that $\mathsf{P}_{\mathsf{search}}$ would be the same if we replace Word-RAM any strongly Turing-equivalent model, like Turing Machines (described below).

By this definition, Shortest Paths, 2-Coloring, Sorting, Interval Scheduling, Bipartite Matching, 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, Independent Set, or Longest Path 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 2.2.
$\mathsf{P}_{\mathsf{search}} \subsetneq \mathsf{EXP}_{\mathsf{search}}$, *and* $\mathsf{P} \subsetneq \mathsf{EXP}$.

We won't give a proof of this theorem (take CS 121 for that), but we'll see similar proofs in the last unit of the course.

We even know (again, without proof) an example of a problem in $\mathsf{EXP}_{\mathsf{search}} \setminus \mathsf{P}_{\mathsf{search}}$ (in fact $\mathsf{EXP} - \mathsf{P}$): the problem of deciding whether a Word-RAM program halts on an input x of length n within 2^n steps, called the "Bounded Halting" problem.

Next we might try to obtain more intractable problems via reductions.

Definition 2.3. 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 should exist $c \in \mathbb{R}$ such that on an input of length N, the reduction runs in time $O(N^c)$, if we count the oracle calls as one time step (as usual).

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

- GraphColoring $\leq_p$ SAT
- LongPath $\leq_p$ SAT

Lemma 2.4. *Let Π and Γ be computational problems such that $\Pi \leq_p \Gamma$. Then:*

1. *If $\Gamma \in \mathsf{P}_{\mathsf{search}}$, then $\Pi \in \mathsf{P}_{\mathsf{search}}$.*
2. *If $\Pi \notin \mathsf{P}_{\mathsf{search}}$, then $\Gamma \notin \mathsf{P}_{\mathsf{search}}$.*

This is the same lemma as we introduced in lecture 3 and recalled yesterday, but keeping track only of whether there are polynomial-time algorithms solving the problems, not more precise runtimes.

Proof.

1. **Reduction:** Since Π reduces in polynomial time to Γ, there is some reduction R solving Π which runs in time $O(n^c)$ for some c on inputs of size n given an oracle that solves Γ.

   Oracle replacement: We assume that Γ is in P, i.e. for some constant d, there is an $O(n^d)$ algorithm A solving Γ on inputs of length n. Then we can create an algorithm that, given an input x to Π, runs R on x. Whenever R calls the oracle on a value y_i, we instead evaluate $A(y_i)$.

   Overall runtime: The overall runtime, including reduction and oracle calls, is

   $$O(n^c + \#\{\text{Calls made to oracle}\} \cdot B)$$

   where B is an upper bound on the size of the oracle calls. Then, since R runs in time $O(n^c)$ and calls the oracle at most once per step, R also calls the oracle at most $O(n^c)$ times. Furthermore, given that R runs in time $O(n^c)$ and writes at most w bits to memory per time step, each oracle call is also on an input of size of at most $O(n^c w)$. Since $w \leq n$ initially (the input has at least one word) and w increases by at most 1 per time step, each oracle call is on an input of size $O(n^c n^c) = O(n^{2c})$.

   This gives a total runtime bound of

   $$O(n^c + n^c(n^{2c})^d) = O(n^{c\cdot(2d+1)}),$$

   which is still polynomial.

2. Contrapositive of 1.

These runtime blowups are acceptable because we are still inside P. If we had defined computationally efficient as e.g. quadratic time, we wouldn't be able to compose reductions in this way. □

So, we have a procedure for proving that problems are not in $\mathsf{P}_{\mathsf{search}}$:

1. Identify a particular problem Π in $\mathsf{EXP}_{\mathsf{search}} \setminus \mathsf{P}_{\mathsf{search}}$. One example is deciding whether a Word-RAM program halts within 2^n steps on an input x of length n.

2. Show that Π reduces to the problems we are interested in, via a *polynomial-time reduction*.

Unfortunately, we don't know how to reduce the problems we know in $\mathsf{EXP}_{\mathsf{search}} \setminus \mathsf{P}_{\mathsf{search}}$ (like Bounded Halting) to many of the problems we care about (like Independent Set, 3-Coloring, and Longest Path), so we can only conjecture that those problems are in $\mathsf{EXP}_{\mathsf{search}} \setminus \mathsf{P}_{\mathsf{search}}$.

So we have many possible worlds:

These problems have additional structure, which will require us to define and study a different complexity class, NP, next time.

Lemma 2.5. *We can compose reductions - if $\Pi \leq_p \Gamma$ and $\Gamma \leq_p \Theta$ then $\Pi \leq_p \Theta$.*

The proof of this is similar to the proof of Lemma 2.4: run a reduction from Π to Γ, but whenever an oracle call to Γ is made, substitute in the reduction from Γ to Θ.

3 Optional reading: Turing Machines

Most courses on the theory of computation (like CS121) use Turing Machines as their main model of computation, whereas we use the (Word-)RAM model because it better suited for measuring the efficiency of algorithms. However, Turing machines can be understood as a small variant of Word-RAM programs, where we make the word size *constant*:

Definition 3.1 (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

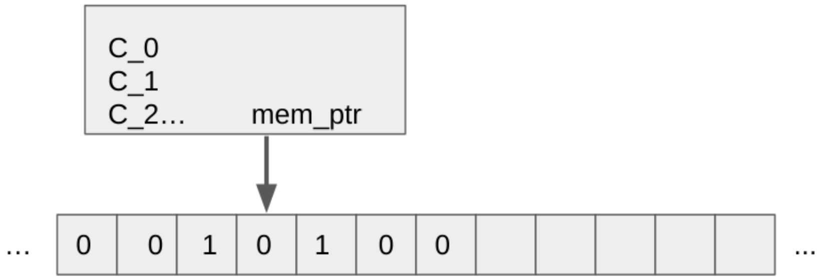

Figure 1: A TM RAM machine, with memory pointer and commands.

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.

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

Thus, the computation can be more concisely described as follows:

Definition 3.2 (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.

Semantics of δ: $\delta(q, \sigma) = (q', \sigma', m)$ means that if the current state is q and $M[$`mem_ptr`$]$ equals σ, then we transition to state q', overwrite $M[$`mem_ptr`$]$ with σ' and increment/decrement/maintain `mem_ptr` according to whether $m = R$ ("move right"), $m = L$ ("move left"), $m = S$ ("stay in place").

Theorem 3.3 (Equivalence of TMs and TM-RAMs)**.**

1. *There is an algorithm that given TM-RAM program P, constructs a Turing Machine M such that $M(x) = P(x)$ for all inputs x and* $\text{Time}_M(x) = O(\text{Time}_P(x))$.
2. *There is an algorithm that given a Turing Machine M, constructs a TM-RAM program P such that $P(x) = M(x)$ for all inputs x and* $\text{Time}_P(x) = O(\text{Time}_M(x))$.

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?

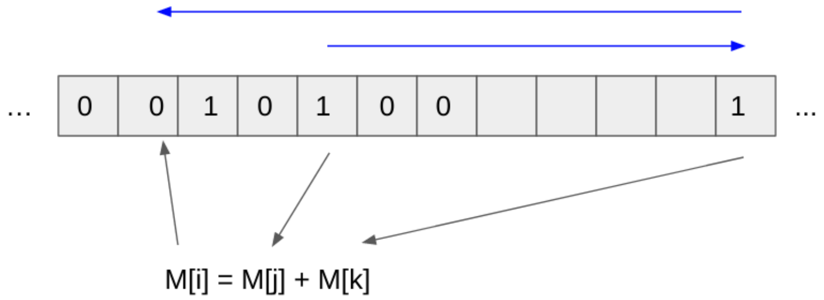

Figure 2: The requirement to move the memory pointer step by step in TM-RAM induces an up to quadratic slowdown vs RAM.

Theorem 3.4. *There is an algorithm that given a Word-RAM Program P constructs a TM-RAM program P' such that $P'(x) = P(x)$ for all inputs x and*

$$\mathrm{Time}_{P'}(x) = O\left((\mathrm{Time}_P(x) \cdot \log \mathrm{Time}_P(x))^2,\right).$$

provided that $\mathrm{Time}_P(x)$ *is at least* $n \cdot \max_i x[i]$ *for an input array* x *of length* n.

Proof Sketch.

- Memory of P: encoded as $S \cdot w$ bits in the memory of P', at a point in the computation when P uses S bits 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 and w, 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 takes time

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

Also, $S \leq T + n = O(T)$ (because each step increases S by at most 1, starting from n) and $w = O(\max\{\log n, \max_i x[i], S\}) = O(\log T)$, so we get time

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

□

So TM-RAMs and Turing Machines can simulate Word-RAM programs, but with a bit more than a quadratic 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 (as demonstrated by your experiments on PS3).