Lecture 5 — Computational models

Ryan's personal draft notes

September 23, 2013

1 Get a scribe — warn him/her that there will be lots of talking, little writing

2 Computational models — why they sometimes matter

Here are two basic algorithms problems:

Palindromes. Given as input $x \in \{0,1\}^n$, is it a palindrome? (I.e., is it $x = ww^R$ for some w?)

Sorting. Given as input n integers, sort them.

What running time is required to solve these problems? The naive answers are $O(n)$ for Palindromes and $O(n \log n)$ for Sorting. Sophisticates might suggest $O(n)$ time for Sorting (e.g., with "Radix Sort"). On the other hand, other sophisticates might know an $\Omega(n \log n)$ *lower bound* for sorting. There are $n!$ different input orders, and any comparison between two elements yields only one "bit" of information; therefore, by a simple information theory argument, $\log(n!) = \Omega(n \log n)$ comparisons are needed. What gives?

In fact, for both Palindromes and Sorting, the answer is:

It depends on the computational model.

Let's briefly discuss Palindromes. The obvious algorithm is:

```
for i = 1 ... n
   if x[i] $\neq$ x[n−i] then return NO
return YES
```

(I don't care about the simplification of letting i only go up to $n/2$; it'll just confuse things.) Looks like $O(n)$ time, right? And indeed, if you implement it on a standard computer with $n = 10^5, 10^6, 10^7, 10^8$, I'm sure you'll find the running time scales linearly. But if you try to justify that "in theory", things arguably get annoying. Writing down `i` takes $\log n$ bits. So does each increment of `i` cost $O(\log n)$ time? Does it really take only time $O(1)$ to access `x[1]` and `x[n]` in the first iteration? (Certainly not on the basic TM you learned about as an undergrad!)

The running time of Sorting is an even more complex issue. In fact as we'll see towards the end of the lecture, it is fair to say "the optimal time complexity of sorting is unknown". (Or at least

— "there are still open aspects to the problem".)

This is a course about TCS, so it makes sense to get our models straight towards the beginning. If you end up working on algorithms — especially highly efficient ones — you'll need to know these details. But even if you work on pure complexity, you still need to know these details. (E.g., what is an NP-hardness result but an algorithm? And there are strong reasons to desire highly efficient NP-hardness reductions. . .)

We're going to talk about three basic models of computation in this lecture:

1. Turing Machines (TMs);
2. Circuits;
3. The Word RAM.

3 Turing Machines

I hope you remember the basics of what a TM looks like and how it works.

picture of a basic one-tape TM goes here

Historical main advantage of TMs: It is extremely clear what their running time / space usage is. (Number of steps till accept/reject state; number of tape cells accessed.)

Unfortunately, it's quite unrealistic; it gives running times that are too slow compared to standard computers. For example, how long does it take to solve Palindromes on a one-tape TM? If you think about it, it seems clear you have to scan back-and-forth and back-and-forth — running time $O(n^2)$. This is the truth.

Theorem 3.1. *(Hennie, 1965.) Solving Palindromes on a one-tape TM requires time* $\Omega(n^2)$.

This just goes to show the "memory access model" of one-tape TMs is kind of ridiculous. One standard "solution" in complexity theory is to look at multitape TMs:

picture of a basic multitape TM goes here; indicate input tape, work tapes, output tape

Especially for space analysis, it's standard to let the input be on a read-only "input tape", let the output be on a write-only "output tape", and call the other tapes "work tapes"; only the space usage on the work tapes "counts".

Remark 3.2. Hennie–Stearns 1966 showed that an $O(1)$-tape machine running in time $T(n)$ can be simulated by a 2-tape machine running in time $O(T(n)\log T(n))$ and a 1-tape machine in time $O(T(n)^2)$.

Fact 3.3. *On a two-tape machine, Palindromes can be solved in* $O(n)$ *time: just copy the input in reverse onto the work tape and check equality.*

Though that looks fine, it might look like "luck"; even two-tape TMs look like they are not modeling "real" computers very accurately. In fact, there's still a deficiency about this result for Palindromes — the solution uses too much space! It uses n space, whereas the standard pseudocode solution given at the beginning of the lecture seems to only use $O(\log n)$ space (to store the index i). Indeed, this is an insuperable problem even for two-tape machines:

Theorem 3.4. *(Apparently Cobham, 1966, according to Duris–Galil '84.) If an $O(1)$-tape TM solves Palindromes with time $T(n)$ and space $S(n)$ then $T(n)S(n) = \Omega(n^2)$.*

You might suggest that even this can be evaded if you allow two tape heads per tape, but Duris–Galil proved similar results even for $O(1)$ tape heads per tape. All of this suggests that despite being a fairly common standard in complexity theory, multitape TMs are not really a great choice for realistic modeling of resources.

These difficulties stem from the limited "memory access model" of Turing Machines, and are alleviated by allowing for "random access memory". To model this, we don't want to get into x86 architecture or anything; we stick with TMs. Here is a standard definition:

Definition 3.5. A *random access memory (RAM) TM* is the usual multitape model except:

- Some of the work tapes can be "random access". Means they have a corresponding "index tape". For each such tape, the TM can do a "jump" instruction, which moves the head on the random access tape to the cell # written (in binary) on the associate index tape. The contents of the index tape are also reset.

The RAM TM should be the standard model for complexity theory (and often is). It can solve Palindromes in $O(n)$ time and $O(\log n)$ space, as desired.

Remark 3.6. Recall the 2007 Williams theorem mentioned last lecture: SAT cannot be solved simultaneously in time $n^{2\cos(\pi/7)-\epsilon}$ and space $n^{o(1)}$ (for any $\epsilon > 0$). Thankfully, Williams indeed showed this for RAM TMs! (Since already Cobham'65 had a much stronger result for multitape TMs!)

Remark 3.7. For hardcore algorithms work, even the RAM TM is not quite realistic enough. For example, consider:

FindMax. Given n positive integers, each less than n, find the maximum.

This "feels like" it should be $O(n)$ time, but actually it requires time $\Omega(n \log n)$ — because even just the *input length* is $n \log n$. Here we're quibbling about $\log n$ factors; complexity theorists probably don't care, but algorithmicists do. More on this at the end of the lecture.

4 Circuits

Circuits: a highly concrete, unambiguous measure of complexity when the input length, n, is *fixed.* Here instead of languages, we look at computing boolean *functions*, $f : \{0,1\}^n \to \{0,1\}^m$ (most often $m = 1$). In some sense, circuits are pretty "realistic"; I mean, computer chips are circuits, more or less.

A circuit looks like this:

*draw a simple depth-*3 *de Morgan circuit*

Definition 4.1. Let $\mathcal{B}$ be a set of "gate functions"; e.g., $\mathcal{B}$ might consist of $\{\neg, \vee, \wedge\}$ (unary, binary, and binary functions). An *n-input circuit C with basis $\mathcal{B}$* is a sequence of $t \geq n$ boolean functions ("gates"), $g_1, \ldots, g_t$. The first n are always the "input variables": $g_1 = x_1$, $g_2 = x_2$, $\ldots$, $g_n = x_n$. Each subsequent g_i is the application of some $\phi \in \mathcal{B}$ to some previous collection of g_j's (each $j < i$).

Note that the g_i's therefore form a *directed acyclic graph (dag).* The in-degree of each gate is called its *fanin.* (Input gates have fanin 0; a ϕ-gate has fanin equal to the arity of ϕ.)

Some sequence $g_{i_1}, \ldots, g_{i_m}$ of gates are designated the *output gates* (most often $m = 1$).

Given circuit C, we identify it with the function it computes, so $C : \{0,1\}^n \to \{0,1\}^m$.

The *size* of a circuit is $t - n$: only the non-input gates count. The *depth* is the length of the longest input-output path in the dag.

Twist: we also let gates use the constants 0 and 1 as inputs "for free".

Definition 4.2. A *formula* is a circuit where all gates have fanout 1; i.e., the dag is actually a tree. Here we usually *do* count the input gates toward the size.

Definition 4.3. Popular basis choices:

- B_2: all 16 2-bit functions.
- U_2: $\{\neg, \vee, \wedge\}$, with fanin-2 AND and OR. These are called *de Morgan circuits.* Twist: traditional to let $\neg$ gates be "free" (w.r.t. size, depth).
- de Morgan circuits except that AND and OR can have *arbitrary* fanin.

Notice that the difference between U_2 and B_2 (or even $B_{O(1)}$) just affects size/depth by constant factors, but allowing arbitrary fanin makes a larger difference. For example, the function AND : $\{0,1\}^n \to \{0,1\}$ has size 1 and depth 1 with arbitrary fanin but requires depth about $\log n$ and size about n over U_2 or B_2 (make a binary tree of ANDs).

An advantage of circuit complexity is it's even easier to be brutally concrete/unambiguous. For example:

Theorem 4.4. *(Meyer–Stockmeyer 1974.) Let WS1S be the problem of deciding if a sentence in a certain weak form of 2nd-order logic over $\mathbb{N}$ is true. Note: WS1S is decidable. Any circuit over B_2 which correctly decides all sentences of 3660 input bits requires size at least 10^{125}.*

4.1 Palindromes

Let's solve Palindromes with circuits. Actually, inputs to Palindromes can be of any length, so what does that mean?

Definition 4.5. A *circuit family* is a sequence of circuits $(C_n)_n$, one for each input length. It solves a problem P if C_n solves $P \cap \{0,1\}^n$.

So how will we solve length-n Palindromes? We just need to check equality of all the pairs (x_1, x_n), (x_2, x_{n-1}), etc., and AND them together.

draw a little OR of AND of x_1, x_n and AND of $\neg x_1, \neg x_n$; then a big AND

Remark 4.6. This has size $n + 1$ as an unbounded fanin circuit. Actually, it's a formula, too. It has depth 3, but can you see how to make it depth 2? Just use CNFs, not DNFs to check equality.... Gives you AND(OR($x_1, \neg x_n$), OR($\neg x_1, x_n$), ...).

That's quite nice/realistic.

4.2 More on circuits

Definition 4.7. Here are the (ridiculous) names of some "circuit classes":

- AC^0: all languages $L \subseteq \{0,1\}^*$ such that there's a constant d such that L is computed by (a family of) unbounded fanin de Morgan circuits of depth d and size $\text{poly}(n)$.
- NC^1: $O(\log n)$-depth, $\text{poly}(n)$-size over U_2. Note: $\mathsf{AC}^0 \subseteq \mathsf{NC}^1$.
- NC: $\text{poly}\log n$-depth, $\text{poly}(n)$-size over U_2 for some $d \in \mathbb{N}$.
- P/poly: $\text{poly}(n)$-size over U_2.

Never mind what "AC" and "NC" stand for.[1]

Remark 4.8. How do circuits compare with TMs? Very roughly, circuit size corresponds to time. For example, you can evaluate a circuit of size $s \geq n$ in time $O(s \log s)$ on a RAM TM. Conversely:

Theorem 4.9. *(Pippenger–Fischer '79): A multitape TM running in time $T(n)$ can be simulated on length-n inputs by a circuit of size $O(T(n) \log T(n))$. [$O(T(n)^2)$ is easy.]*

And *very roughly*, depth corresponds to *parallel time* — imagine being able to put a processor on each gate. E.g., any problem not in NC probably doesn't have a highly efficient parallel algorithm.

Given the above it may seem like polynomial time and polynomial-size circuits are the same. There is one catch, though, illustrated by the following:

Fact 4.10. *The Halting Problem can be solved by circuits. Why? Because the Halting Problem restricted to n-bit inputs is just some function $f : \{0,1\}^n \to \{0,1\}$, and for every function there exists a U_2-circuit of size $n2^n$ computing it. (In fact, Shannon '49 gave an easy argument for $O(2^n/n)$.)*

This fact illustrates something slightly ridiculous about circuits, which is that you get to reason existentially about each input length. You don't have to worry about the *time it takes to construct the circuit*. This issue is called **non-uniformity**. One can alleviate it by requiring "uniformity":

Definition 4.11. A circuit family (C_n) is called *uniform* if there is an algorithm (TM) which on input "n", outputs a description of C_n in $\text{poly}(n)$ time.

Remark 4.12. The set of languages decided by uniform, polynomial-size circuit families is the same as those decided by polynomial time TM algorithms.

In the '80s people thought maybe you would show $\mathsf{P} \neq \mathsf{NP}$ by showing that even $\mathsf{NP} \not\subseteq \mathsf{P}/\text{poly}$; i.e., by showing SAT does not have poly-size circuits. This plan does not look too hopeful. On one hand:

Proposition 4.13. *There exist functions $f : \{0,1\}^n \to \{0,1\}$ which require circuits of size $\Omega(2^n/n)$.*

Proof. It's easy to estimate that the number of distinct size-s circuits is $s^{O(s)}$. The number of n-bit functions is 2^{2^n}. The former is smaller unless $s = \Omega(2^n/n)$. $\square$

On the other hand:

[1] Alternating Circuits and Nick's Class, respectively.

Fact 4.14. *The best circuit-size lower bound known for an "explicit"*[2] *family of functions is:*

- *$3n - o(n)$ for basis B_2 [Norbert Blum '84];*
- *$5n - o(n)$ for basis U_2 [Iwama–Morizumi '02, building on Lachish–Raz '01].*

4.3 Sorting with circuits

We looked at Palindromes in the circuit model — what about Sorting? As mentioned before, there is the slightly awkward fact that it seems like Sorting n integers may have input size $\Omega(n \log n)$. So let's in fact look at a simpler problem: sorting *bits*. If you think about it for a second, you see that sorting n bits just means counting the number of 1's in the input and then outputting this count in unary. Here are some slightly simpler problems:

Definition 4.15.

Majority : $\{0,1\}^n \to \{0,1\}$. (I.e., output the middle bit of the unary count.)

Parity : $\{0,1\}^n \to \{0,1\}$. (I.e., output the OR of the odd bits of the unary count.)

Output the number of 1's in the input in *binary.* (If you can do this, you can do the other two, by outputting the MSB/LSB, respectively.)

In fact it's not hard to see you can do the last of these with size $O(n)$. Either left as an exercise, or here: "Do it recursively; split the input into two blocks, count, add up the two $\log n$ bits numbers. The recursion is $S(n) = 2S(n/2) + O(\log n)$, which solves to $O(n)$." You can also do it in log-depth; this is a bit trickier to account for.

Here we implicitly considered circuits for addition. It's easy to see that adding two m-bit numbers can be done in $O(1)$-depth and $O(n)$ size in the unbounded fanin model.

Oddly enough, here is a famous circuit lower bound:

Theorem 4.16. *Majority, Parity, counting-in-binary, and Multiplication of two numbers cannot be computed in AC^0. (First proved by Furst–Saxe–Sipser '81, the sharpest version due to Håstad.)*

5 The Word RAM model

It's complexity theorists that mainly care about circuits, due to their extreme concreteness. Actual algorithmicists would rather work in a model that's closer to "a programming language, as interpreted by an actual compiler on actual hardware". Here we return to the issue of, say, adding up n integers between 0 and $n-1$. This seems like it should take $O(n)$ time. Can't we just model, like, basic "C" or "assembly language" programming? Call those integers `int`'s, and say you can add two of them in $O(1)$ time?

Sure, but things get slightly funny. You want to allow one integer to be stored in a "word" of memory. Let's say a "word" has w bits. Well, if $w = O(1)$ then it's not really different (up to constant factors) then having single-bit words. Here's another funny observation: you typically think of *indexing* into the array of n words/integers with a single word ("i"). Doesn't that imply that we need $w \geq \log n$?

[2] "Explicit" is usually taken to mean "in NP", although: a) the lower bounds we state here are for function families in P; b) we don't know better lower bounds even for function families computable by exponential time algorithms with access to an NP oracle.

Answer: Yes! And that's a standard assumption! The first time you see this it may seem totally weird: you're assuming the computer hardware size depends on the input size?! That seems to make no sense. But once you calm down, it's actually quite logical and cool. Of *course* you want a single pointer to fit into a word. You should just think of w as a parameter of the model, and $w \geq \log n$ as an assumption.

Here's what is considered to be the best "standard model" for algorithms results:

Definition 5.1. The *Word RAM model*:

- Memory is divided into *words* of length w bits. For size-n inputs, we always assume $w \geq \log n$.
- It costs time 1 to do "basic" operations on w bits. These always include: addition; subtraction;[3] bitwise AND, OR, NOT; "unlimited" cyclic shift left or right (with zero-padding), meaning by any number of bits at most w.
- What about multiplication, you say? We'll come to that...
- One can do "assembly+RAM language"-style instructions: conditional jumps, indirect memory access, etc. Note that if $w \gg \log n$, you can theoretically do memory access into superpolynomially many words. This is considered a bit tacky; usually one also insists that the machine has at most $S \leq 2^w$ words, and for basic problems like Sorting or Shortest-Paths one wants $S = O(n)$.

Reference: Hagerup, "Sorting and searching on the Word RAM".

Remark 5.2. I personally find it natural to assume that we also have $w = O(\log n)$.[4] I don't really see that it makes much sense to allow storing superpolynomial-sized integers in a single word, and doing unit-cost operations on them. Still, quite a few people are interested in allowing larger w; when you allow for arbitrary $w = w(n) \geq \log n$ and try to get a running time which is independent of w, this is called (believe it or not) the *Transdichotomous Word RAM model.*[5]

Remark 5.3. What about multiplication? On one hand, it seems basic and occurs in most programming languages/architectures. On the other hand, note that the other basic operations (addition, AND, cyclic shift, etc.) are implementable in AC^0 (meaning by poly(w)-size, constant-depth unbounded fanin circuits). Whereas multiplication provably isn't.

For algorithms that are not hyper-concerned about efficiency, and have arithmetic as key components, people usually don't mind if you allow yourself unit-time multiplication (two w-bit words get multiplied, output into two w-bit words).

On the other hand, if you're more hard-core, you work in the AC^0-Word RAM model, where the only allowed unit-cost instructions are the AC^0 ones. (This is sometimes called the *Practical RAM model.*)

So now the problem of adding up n numbers, each of which fits into $O(1)$ words, indeed takes $O(n)$ time. Finally, what about Sorting?

[3] both mod 2^w

[4] Note that constant factors on w don't make any real difference; you can do, e.g., add two 5-word numbers in $O(1)$ time still. So we can allow intermediate results that are $O(1)$ words.

[5] There's also some ridiculousness wherein you have to allow the algorithm to know certain constants related to w; e.g., w. These can be computed in $O(\log w)$ time, which is usually negligible, but technically might not be if w is enormous. I don't want to elaborate further.

5.1 Sorting

Let's assume for a while that $w = \Theta(\log n)$.

Suppose we want to sort n integers in the range $0 \dots n-1$. How can we easily do this in $O(n)$ time?

Counting sort: Allocate an array for each key (size: n words), and count the number of times each integer occurs. Then you can easily reconstruct the sorted array.

Unfortunately, this doesn't work very well if the range is, say, $0 \dots n^{50} - 1$ (each representable using 50 words of length $\log n$, or 1 word of length $50 \log n$). Why not? (It uses space n^{50}, and also that much time, I think). Instead, there is...

Radix sort: Suppose you sort all the numbers first by *least significant bit*, then by *second-least significant bit*, etc., up to *most significant bit.* Then assuming you are using a "stable sort" (two numbers that are tied never get their positions reversed), this actual sorts the numbers. (Takes a minute to think about why.) Note that here we are sorting bits, so we can easily do Counting Sort. This looks like it will take time $O(nw)$.

However, one could then imagine sorting *byte*-wise, rather than *bit*-wise. This can also be done by counting sort (needs an array of size 2^8), and saves you a factor of 8. You can then go up to a radix of k-bits, for $k \le w$; you'll need extra space 2^k, and it'll take time $O(nw/k)$. Consequence:

Theorem 5.4. *Radix Sort can sort n words using $O(n)$ space and $O(\frac{n}{\log n} w)$ time. For the standard setting of $w = O(\log n)$, this is $O(n)$ time.*

Great! As you can see, though, if you are hard-core and care about $w \gg \log n$ then Radix Sort is actually terrible; for $w \ge \log^2 n$ you may as well just use a classic $n \log n$ comparison-based sorting algorithm. Still, there are improvements for large w (all of the following use linear space).

- Using the classic "van Emde Boas priority queue" (1974): $O(n \log w)$ time. This is $O(n \log \log n)$ time whenever $w = \text{polylog}(n)$ (and I think larger w is kind of ridiculous). Also can be improved to $O(n \log(\frac{w}{\log n}))$ to match Radix [Kirkpatrick–Reisch '84].
- Fredman–Willard '90: $O(n \frac{\log n}{\log \log n})$ for all w. (Heavily uses multiplication.)
- Signature Sort [Andersson–Hagerup–Nilsson–Raman '95]: $O(n)$ time (randomized, no multiplication) whenever $w \ge \log^{2+\epsilon} n$.
- Thorup: an equivalence between sorting and $n\times$ priority queues.
- Han '04: Deterministic $O(n \log \log n)$ time for all w, no multiplication.
- Han–Thorup '02: Randomized $O(n\sqrt{\log \log n})$ time, uses multiplication. Also improvable to $O(n\sqrt{\log \frac{w}{\log n}})$

Thus in a sense (the transdichotomous model), whether or not we can sort in $O(n)$ time is open.

5.2 Dictionary problem

Here we want a data structure that maintains a set of integers (numbers between 0 and $2^w - 1$) and supports Insert, Delete, Successor, Predecessor.

- Balanced binary tree (classic solution): $O(\log n)$ time/op, $O(n)$ space.
- van Emde Boas tree [1974]: $O(\log w)$ time/op, $O(2^w)$ space (not great).
- y-fast trees [Willard '83]: $O(\log w)$ time/op whp, $O(n)$ space.
- Fusion trees [Fredman–Willard '93]: $O(\frac{\log n}{\log w})$ time/op, $O(n)$ space, AC^0 only.

5.3 Single-source shortest paths

[Thorup '99] $O(m)$ time in the undirected case. (Uses multiplication, have to throw in Ackermann factor if you want to ditch it.)

6 Multiplication, and other arithmetic

Finally, let's come back to multiplication. Forget multiplying "words"; let's just try to get a bead on multiplying two n-bit numbers. For simplicity now, let's work in either the RAM TM or in the U_2-circuit model. How long does it take? You may know:

- Grade school: $O(n^2)$ time.
- (Karatsuba '62:) Using divide-and-conquer (Strassen-like): $O(n^{\log_2 3}) = O(n^{1.585})$.
- (Toomey-Cook, mid-'60s) — a souped version: for each $\epsilon > 0$, time $f(\epsilon) \cdot n^{1+\epsilon}$ where $f(\epsilon) \to \infty$ (quickly) as $\epsilon \to 0$.
- (Schönhage-Strassen algorithm, 1971): $O(n \log n \log\log n)$ using FFT.

At this point, we're into the realm where logs matter. In fact, if we now go to the Word RAM model...

Corollary 6.1. *(Appears in Knuth TAOCP2.) In the Word RAM model with $w = \log n$ and multiplication (of w-bit words) allowed, two n-bits numbers can be multiplied in $O(n)$ time.*

38 years after the SS algorithm, there was a startling improvement:

Theorem 6.2. *(Fürer 2009): In the circuit/multitape-TM models, two n-bit integers can be multiplied in time $O(n \log n 2^{\log^* n})$.*

(Recall: $\log^* n$ is the number of times you need to take log to get down to 2 from n.)

6.1 Other arithmetic

The reference for this entire section is the Brent–Zimmermann book "Modern Computer Arithmetic", 2010.

What about... division? Now you start to enter the realm of questions about *real numbers*. The simplest way to talk about floats is to say that they are represented in the form $m2^e$ where $m, e \in \mathbb{Z}$.

Goal: get n bits of precision for m in output, given n bits of precision in input.

Definition 6.3. Let $M(n)$ denote the time two multiply two n-bit integers. We know $M(n) = O(n \log n 2^{\log^* n})$ in the circuit/TM models.

Known facts:

- Time to divide one n-bit integer by another: $O(M(n))$. Technique: find $1/y$ via Newton's method, solving $xy - 1 = 0$. (By adjusting e, can nail down m to a factor of 2. Then to get a close starting point for Newton, use lookup tables.... These lookup tables being wrong were the source of the Pentium division bug!)
- Square-root, kth root: time $O(M(n))$. Technique: Newton's method.
- (Extended) GCD: $O(M(n) \log n)$. Technique: NOT Euclid's algorithm. Knuth was the first to get $\widetilde{O}(n)$.
- Base conversion: $O(M(n) \log n)$.
- ln: $O(M(n) \log n)$. Technique: Arithmetic-Geometric Mean iteration.
- exp: $O(M(n) \log n)$. Technique: inverse of ln, using Newton's method.
- "Any" elementary functions (e.g., sin): also $O(M(n) \log n)$, also based off of doing ln.
- Computing π, or e: $O(M(n) \log n)$, also based of ln and other techniques.