CS1200: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Lecture 7: RAM Simulations and the Church-Turing Thesis

Harvard SEAS - Fall 2025 *2025-09-23*

1 Announcements

Recommended Reading:

- Hesterberg–Vadhan Ch. 6
- MacCormick Ch. 5

In the previous class, we introduced the RAM model of computation, convinced ourselves that it is unambiguous and mathematically simple, and started to convince ourselves of its expressivity—for instance, we can sort in time $O(n^2)$ in the RAM model. Our remaining desiderata for a computational model are expressivity (can we actually program everything we intuitively consider to be an algorithm?), robustness (would small tweaks to the model have made a fundamental difference?), and technological relevance. In this chapter, we'll show via *simulation arguments* that the RAM model satisfies all three of those desiderata.

2 Expressiveness: Simulating High-Level Programs

We claim that, despite its simplicity, the RAM model is extremely powerful; it is *equivalent* in expressiveness to all of our familiar high-level programming languages.

Theorem .1 (informal)**.** [1]

1. *Every Python program (and C program, Java program, OCaml program, etc.) can be simulated by a RAM Program.*
2. *Conversely, every RAM program can be simulated by a Python program (and C program, Java program, OCaml program, etc.).*

Q: What do we mean by "simulation"?

[1]This is only an informal theorem because some of these high-level programming languages have fixed bounds on the size of numbers or the size of memory, whereas the RAM model has no such constraint. To make the theorem correct, one must work with generalizations of those languages that allow for varying word and memory sizes, similar to the Word-RAM Model we introduce in Chapter **??**.

A:

Proof Idea. 1.

2.

□

Data encodings in simulations.

Efficiency of simulations.

3 Robustness

We made somewhat arbitrary choices about what operations to include or not include in the RAM Model—roughly, we picked a set of operations sufficient to be expressive, but a minimal such set, to optimize the mathematical simplicity criterion. However, we could easily have picked a slightly different minimal set of operations that's equally expressive. Alternatively, often a wider set of operations is allowed, both in theoretical variants of the RAM model and in real-life assembly language.

It turns out that the choice of operations does not much affect what can be computed. Specifically, we establish robustness of our model by *simulation theorems* like the following:

Theorem .2. *Define the* mod-extended RAM model *to be like the RAM model, but where we also allow a mod (%) operation. Then every mod-extended RAM program P can be simulated by a standard RAM program Q. Moreover, on every input x, the runtime of Q on x is at most 3 times the runtime of P on x.*

Corollary .3. *A problem is solvable in time $O(T(n))$ by a RAM program if and only if it is solvable in time $O(T(n))$ by a mod-extended RAM program.*

Proof.

□

The constant-factor blow up of 3 can be absorbed in $O(\cdot)$ notation, so we have

$$T_Q(x) = O\left(T_P(x)\right),$$

as desired. Thus, the choice of whether or not to include the mod operation does not affect the asymptotic growth rate of the runtime.

4 Technological Relevance

In Theorem .1, we argued that any program we execute on our physical computers can be simulated on the RAM Model, since the RAM Model can simulate assembly language. However, in the converse direction, we showed only that any RAM program can be simulated by an idealized version of Python that can handle arbitrarily large numbers, leaving open the possibility that the RAM Model is *too powerful* and makes problems seem easier to solve than is possible in practice.

Two issues come to mind:

1.

2.

Addressing Issue 2 requires a new model, which we introduce in the next chapter.

We address Issue 1 via another simulation theorem:

Theorem .4. *There is a fixed constant c such that every RAM Program P can be simulated by a RAM program P' that uses at most c variables. (Our proof will have $c \leq 8$ but is not optimized to minimize c.) Moreover, for every input x,*

$$T_{P'}(x) = O\left(T_P(x) + |P(x)|\right),$$

where $|P(x)|$ denotes the length of P's output on x, measured in memory locations.

Three levels of describing algorithms:

- Low-level: formal code in a precise model like RAM.

- Implementation-level: describing how the memory is laid out in the RAM program, how it uses its variables, and the general structure of the program.
- High-level: mathematical pseudocode or prose.

In most of CS1200 we use high-level descriptions, but in this unit we want to learn how they translate to implementation-level and low-level ones that are actually executed on our computers.

Proof. For starters, we modify P so that its output locations never overlap with its input locations. That is, whenever P halts, we have $\texttt{output\_ptr} \geq \texttt{input\_len}$. We can modify P to have this property by

This modification increases the runtime of P by at most $O(\texttt{output\_len}) = O(|P(x)|)$.

Now, suppose that P has v variables $\texttt{var}_0, \texttt{var}_1, \ldots, \texttt{var}_{v-1}$, numbered so that $\texttt{var}_0 = \texttt{input\_len}$. In the simulating program P', we will instead store the values of these variables in memory locations

$$M'[\texttt{input\_len}'], M'[\texttt{input\_len}' + 1], \ldots, M'[\texttt{input\_len}' + v - 1],$$

where we write $\texttt{input\_len}'$ to denote the input-length variable of P' to avoid confusion with variable $\texttt{input\_len}$ of P. For other memory locations, $M[i]$ will be represented by $M'[i]$ if $i < \texttt{input\_len}'$, and $M[i]$ will be represented by $M'[i+v]$ if $i \geq \texttt{input\_len}'$.

Now, to obtain the actual program P' from P, we make the following modifications.

1. Initialization: we add the following line at the beginning of P'

2. A line in P of the form $\texttt{var}_i = \texttt{var}_j\ \texttt{op}\ \texttt{var}_k$ can be simulated with $O(1)$ lines of P' as follows:

3. A conditional $\texttt{IF var}_i == 0\ \texttt{GOTO}\ k$ can be similarly replaced with $O(1)$ lines of code ending in a line of the form $\texttt{IF temp}_2 == 0\ \texttt{GOTO}\ k'$. (Note that we will need to change the line numbers in the GOTO commands due to the various lines that we are inserting throughout.)

4. Lines where P reads and writes from M are slightly more tricky, because we need to shift pointers outside the input region by v to account for the locations where M' is storing the variables of P. Specifically, we can replace a line $\texttt{var}_i = M[\texttt{var}_j]$ with a code block that does the following:

Again, one line of P has been replaced with $O(1)$ lines of P'.

5. Writes to memory are handled similarly to reads.

6. Setting output: set the variables `output_len`$'$ and `output_ptr`$'$, by reading the memory locations corresponding to the variables $\texttt{var}_i$ = `output_len` and $\texttt{var}_j$ = `output_ptr`, and incrementing the output pointer by v as above. (This is where we use the assumption that the output of P is always in memory locations after the input.)

All in all, we have replaced each line of P by $O(1)$ non-looping lines in P'. Thus we incur only a constant-factor slowdown in runtime (on top of the additive $O(|P(x)|)$ slowdown we may have incurred in the initial modifications of P), and our new program only uses $O(1)$ variables: $\texttt{temp}_0$ through $\texttt{temp}_4$, `input_len`$'$, etc.—none of the variables $\texttt{var}_i$ is a variable of our new program. □

5 The Church–Turing Thesis: Simulating the Universe

As a consequence of Theorems .1 and .2, and expanding it to include a few other models, we see that many different models of computation are equivalent in power, meaning that they can solve the same class of computational problems. We refer to such models of computation as *Turing-equivalent* models.

Theorem .5 (Turing-equivalent models). *If a computational problem* Π *is solvable in one of the following models of computation, then it is solvable in all of them:*

The Church–Turing Thesis: The above equivalence of many disparate models of computation leads to the Church–Turing Thesis, which has (at least) two different variants:

1.

2.

This is not a precise mathematical claim, and thus cannot be formally proven, but it has stood the test of time very well, even in the face of novel technologies like quantum computers (which have yet to be built in a scalable fashion); every problem that can be solved by a quantum algorithm can also be solved by a RAM program, albeit much more slowly.

Simple and elegant models: