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

Lecture 7: RAM Simulations

Harvard SEAS - Fall 2024 *Sept. 24, 2024*

1 Announcements

- Revision videos and Regrade requests: Due Sunday, Oct. 1

Recommended Reading:

- CLRS Sec 2.2

2 Loose Ends and Recap

2.1 RAM Semantics

Definition 2.1 (Computation of a RAM Program: semantics)**.** A RAM Program $P = (V, (C_0, \ldots, C_{\ell-1}))$ *computes* on an input x is as follows:

1. Initialization:

2. Execution:

3. Output:

The *running time* of P on input x, denoted $T_P(x)$, is defined to be:

Last time we introduced the RAM Model of Computation, and convinced ourselves that it is unambiguous and mathematically simple, and started to convince ourselves of its expressiveness. Today we complete our discussion of expressiveness and then turn to the desiderata of robustness and technological relevance.

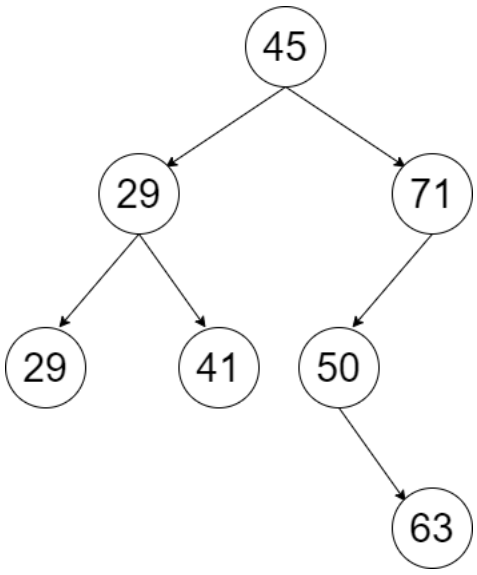
Theorem 2.2. *SortingOnNaturalNumbers can be done in time $O(n^2)$ in the RAM Model.*

2.2 Data Representation

Implicit in the expressivity requirement is that we can describe the inputs and outputs of algorithms in the model. In the RAM model, all inputs and outputs are arrays of natural numbers. How can we represent other types of data?

- (Signed) integers:
- Rational numbers:
- Real numbers:
- Strings:

What about a fancy data structure like a binary search tree? We can represent a BST as an array of 4-tuples (K_0, V_0, P_L, R_R) where P_L and P_R are pointers to the left and right children. Let's consider the example from last class:



Assuming all of the associated values are 0, this would be represented as the following array of length 28:

$$[45, 0, 4, 8, 29, 0, 12, 16, 71, 0, 29, 0, 29, 0, 0, 0, 41, 0, 0, 0, 50, 0, 0, 24, 63, 0, 0, 0]$$

For nodes that do not have a left or right child, we assign the value of 0 to P_L or P_R. Assigning the pointer value to 0 does not refer to the value at the memory location of 0, as we assume that the root of the tree cannot be a child for any of the nodes in the rest of the tree. Note that there are many ways to construct a binary tree using this array representation.

3 Expressiveness: Simulating High-Level Programs

Theorem 3.1 (informal)**.** [1]

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

[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 a generalization of those languages that allows for a varying word size and memory size, similarly to the Word-RAM Model we introduce below in Section **??**.

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"?

A: *Simulating* one computational model $\mathcal{M}$ by another computational model $\mathcal{N}$ is a very common technique in both the theory and practice of computer science. This means that for every program P in model $\mathcal{M}$, there is a program Q in model $\mathcal{N}$ that is *equivalent* to P. Here *equivalence* means that for every input x, $P(x)$ has the "same behavior" as $Q(x)$, meaning that one program halts iff the other halts, and if they do halt, they halt with the same output.

Proof Idea. 1. Compilers! Our computers are not built to directly run programs in high-level programming languages like Python. Rather, high-level programs are *compiled* into Assembly Language, which is then (fairly directly) translated into machine code that is run by the CPU.[2] Our RAM Model is very similar to and can easily simulate assembly language. (In Assembly Language, what we are calling *variables* are referred to as *registers*, and these represent actual physical storage locations in the CPU.) There are some commands available in Assembly Language that we have not included in our RAM Model, but these can easily be handled by additional simulations like those given in Section 4 below.)

2. This is demonstrated in Problem Set 3. Intuitively, Python can store arbitrarily large arrays with arbitrarily large integers, and can emulate all of the given commands allowed in the RAM model. The only command that is not directly supported in Python is GOTO, but that can be simulated using a loop with if-then statements. In Problem Set 3, you will actually construct a *universal* RAM simulator in Python. That is, a Python program U such that for every RAM Program P and input x, $U(P, x)$ halts iff $P(x)$ halts, and if they halt, then the output $U(P, x)$ equals $P(x)$. A *universal simulation* implies a standard simulation, since for every RAM program P, we can hardcode P into U to obtain a Python program $Q(\cdot) = U(P, \cdot)$ that simulates P.

□

Data encodings in simulations. For different computational models, inputs and outputs may be encoded differently (e.g. as bits vs. as arrays of numbers vs. text on `stdin` and `stdout`). Thus, specifying a simulation technically also requires specifying a mapping $\text{enc}_{\mathcal{M}\to\mathcal{N}}$ that takes data z encoded in the format of model $\mathcal{M}$ and gives an re-encoding $\text{enc}_{\mathcal{M}\to\mathcal{N}}(z)$ of z into the format of model $\mathcal{N}$. Then a more precise statement of equivalence says that for every $\mathcal{M}$-encoded input x, $Q(\text{enc}_{\mathcal{M}\to\mathcal{N}}(x))$ equals $\text{enc}_{\mathcal{M}\to\mathcal{N}}(P(x))$. We will often brush over this detail of encodings in our presentation of simulation results, but it is something to keep in the back of our minds.

Efficiency of simulations. Another issue we will care about below, but is not addressed in the theorem above, is the *efficiency* of the simulation, which allows us to relate complexity measures in the two models. That is, if our chosen complexity measure for model $\mathcal{M}$ says that program P

[2] Python is an interpreted rather than compiled language. Python programs (or rather their bytecode) are actually executed by an interpreter that was originally written in C (or Java) but is compiled to assembly language in order to run.

has running time $T(n)$, can we say that the simulating program Q has running time close to $T(n)$ in our complexity measure for model $\mathcal{N}$?

4 Robustness

We made somewhat arbitrary choices about what operations to include or not include in the RAM Model, mainly with an eye to the mathematical simplicity criterion. However, 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 affect what can be computed too much. Specifically, we establish robustness of our model by *simulation theorems* like the following:

Theorem 4.1. *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.*

Proof. Suppose we have an operation in our extended RAM model of the form $\texttt{var}_0 = \texttt{var}_1 \% \texttt{var}_2$. Instead, introduce a new temp variable `temp`, and replace this line of code with:
$\texttt{temp} = \texttt{var}_1 / \texttt{var}_2$
$\texttt{temp} = \texttt{temp} * \texttt{var}_2$
$\texttt{var}_0 = \texttt{var}_1 - \texttt{temp}$

□

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.

5 Technological Relevance

Last time, 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, this leaves 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. Our CPUs only have a fixed number of registers (e.g. 16 registers in an Intel Core i7 processor), whereas the RAM programs allow an arbitrary constant number of registers.

2. The values stored in registers and in memory are not arbitrary natural numbers but are limited by the *word length* (e.g. 64 bits on a 64-bit machine).

We address Issue 1 via another simulation theorem:

Theorem 5.1. *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.) 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.

To avoid getting bogged down in tedious details in this proof, here we will not describe the construction with all the formal details of RAM code, but give an *implementation-level description*, describing how the memory is laid out in the RAM program, how it uses its variables, and the general structure of the program. Implementation-level descriptions of algorithms are intermediate between *low-level descriptions* (where formal code in a precise model like RAM is given) and *high-level descriptions* (like mathematical pseudocode or prose). In most of CS1200, we work with high-level descriptions, but one of the points of this portion of the course is to learn how these high-level descriptions translate into implementation-level and low-level ones that are actually executed on our computers. Keeping that in mind can help us ensure that we analyze runtime correctly even when working with a high-level description.

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 adding a loop at the end of the program that copies the output to locations $\texttt{input\_len}, \texttt{input\_len} + 1, \ldots, \texttt{input\_len} + \texttt{output\_len} - 1$ and sets $\texttt{output\_ptr} = \texttt{input\_len}$. 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' to initialize the variable $\texttt{input\_len}$ of P correctly:
$$M'[\texttt{input\_len}'] = \texttt{input\_len}'$$

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:
   (a) Calculate $\texttt{input\_len}' + j$, by assigning another temporary variable $\texttt{temp}_0$ the value j and then performing the operation $\texttt{temp}_1 = \texttt{input\_len}' + \texttt{temp}_0$.
   (b) Read $\texttt{temp}_2 = M'[\texttt{temp}_1]$ to obtain the value of $\texttt{var}_j$.
   (c) Similarly (using three lines of code) obtain $\texttt{temp}_3 = M'[\texttt{input\_len}' + k]$ for the value of $\texttt{var}_k$;
   (d) Calculate $\texttt{temp}_4 = \texttt{temp}_2 \ \texttt{op} \ \texttt{temp}_3$.
   (e) Similarly to the above, calculate $\texttt{input\_len}' + i$ in $\texttt{temp}_1$ and write $M'[\texttt{temp}_1] = \texttt{temp}_4$.

The above takes $O(1)$ lines of (non-looping) RAM code in P'.

3. A conditional `IF` $\texttt{var}_i == 0$ `GOTO` k can be similarly replaced with $O(1)$ lines of code ending in a line of the form `IF` $\texttt{temp}_2 == 0$ `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:

   (a) Read the value of $\texttt{var}_j$ from $M'[\texttt{input\_len} + j]$ into a temporary variable $\texttt{temp}_2$ using $O(1)$ steps, like in the other operations above.

   (b) If $\texttt{temp}_2 \geq \texttt{input\_len}'$, then add v to $\texttt{temp}_2$. (This is the pointer shifting due to P' using v memory locations for the variables of P.)

   (c) Obtain $M[\texttt{var}_j]$ by reading $\texttt{temp}_3 = M'[\texttt{temp}_2]$.

   (d) Store $\texttt{temp}_3$ as $\texttt{var}_i$ by writing to $M'[\texttt{input\_len}' + i]$ using $O(1)$ steps like in the other operations above.

   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 $\texttt{output\_len}'$ and $\texttt{output\_ptr}'$, by reading the memory locations corresponding to the variables $\texttt{var}_i = \texttt{output\_len}$ and $\texttt{var}_j = \texttt{output\_ptr}$, 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 $\texttt{temp}_4$, $\texttt{input\_len}'$, etc.—none of the variables $\texttt{var}_i$ is a variable of our new program. □

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