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

Lecture 7: RAM and Word-RAM Simulations

Harvard SEAS - Fall 2022 *Sept. 22, 2022*

1 Announcements

- Add/drop 10/3
- Midterm in class 10/4
- New office hours on Google calendar
- Adam OH after class here in Cambridge for quick post-lecture questions
- Salil OH walking from class to his office (SEC 3.327)

Recommended Reading:

- CLRS Sec 2.2

2 Recap

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 expressivenesss. Today we complete our discussion of expressiveness and then turn to the desiderata of robustness and technological relevance.

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

Proof Idea. 1.

[1] This is only an informal theorem because some of these high-level programming languages have fixed word or memory size bounds, 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 growing word or memory size, similarly to the Word-RAM Model we introduce below.

2.

□

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, for every mod-extended RAM program P, there is a standard RAM program P' such that for every input x, P' halts on x iff P halts on x, and if they halt, then $P'(x) = P(x)$ and the runtime of P' on x is at most 3 times the runtime of P on x.*

Proof.

□

The constant-factor blow up of 3 can be absorbed in $O(\cdot)$ notation. 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.

2.

We address Issue 1 via another simulation theorem:

Theorem 5.1. *There is a fixed constant c such that every RAM Program can be simulated by one that uses at most c variables. (Our proof will have $c \leq 8$ but is not optimized.) That is, for every RAM Program P, there is a RAM Program P' that uses at most c variables such that P' halts on x iff P halts on x, and if they halt, then $P'(x) = P(x)$ and*

$$\text{Time}_{P'}(x) = O\left(\text{Time}_P(x) + |P(x)|\right),$$

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 given an *implemention-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 CS120, 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

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

Now, suppose that P has v variables $\texttt{var}_0, \texttt{var}_1, \ldots, \texttt{var}_v$, 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 P's variable $\texttt{input\_len}$. 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.

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

- A line in P 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 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.)

- 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'.

- Writes to memory are handled similarly to reads.
- 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 $\texttt{var}_j =$, 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 section.

6 The Word-RAM Model

As noted above, an unrealistic feature of the RAM Model as we've defined it is it allows an algorithm to access and do arithmetic on arbitrarily large integers in one time step. In practice, the numbers stored in the registers of CPUs are of a modestly bounded *word length* w, e.g. $w = 64$ bits.

Q: how to represent and compute on larger numbers (e.g. multiplying two 1024-bit prime numbers when generating keys for the RSA public-key cryptosystem)?

A:

Q: What's the problem with restricting our RAM Model to only hold w-bit numbers for a fixed constant w (like $w = 64$) and defining all operations to operate on and produce w-bit numbers?

A:

Definition 6.1. The *Word RAM Model* is defined like the RAM Model except that it has a dynamic *word length* w and *memory size* S that are used as follows:

- Memory:
- Operations:

- Initial settings:
- Increasing S and w:

The current values of the word length and memory size are also made available to the algorithm in read-only variables `word_len` and `mem_size`.

In many algorithms texts, you'll see the word size constrained to be $O(\log n)$, where n is the length of the input. This is because most of the algorithms being studied run in time $\text{poly}(n)$. Thus they can access at most $S = \text{poly}(n)$ memory locations, and so a word size of $\log S = O(\log n)$ is sufficient to access all memory. However, in CS120, we will sometimes study algorithms that run in exponential time or don't even halt, and thus may also use much more than $\text{poly}(n)$ memory locations.

A mental model for the dynamically increasing word size: suppose you are running a really long program on your computer, and its memory usage starts to approach the maximum supported by the word size (e.g. 2^{64}); then you can pause the computation, go out buy a new piece of hardware (e.g. a 128-bit machine), transfer your code over, and continue the computation.

Aside from this RAM Model unit in CS120, you usually won't need to worry too much about the distinction between the RAM Model and the Word RAM Model, since the numbers involved and memory usage of our algorithms will typically be polynomial in $\max\{n, x[0], \ldots, x[n-1]\}$, so can all be managed with only a constant-factor increase in word length from its initial setting in Definition 6.1. But it's worth keeping in the back of your mind: if it looks like your algorithm might construct numbers that are much larger than those in the input, then we need to pay closer attention and not treat arithmetic operations as constant time.

Different algorithms researchers allow different sets of basic operations in the Word-RAM Model, just like different CPUs have different instruction sets. Some of these make only a constant-factor difference in running time (like the % example we discussed earlier), but some may make a difference that depends polynomially on the word length w. For example, it is not obvious how to implement the bitwise XOR of two w-bit words using a constant number of operations in the Word-RAM model we defined, but it can be done using $O(w)$ operations, which usually translates to a logarithmic factor in runtime.[2] It turns out that allowing bitwise operations, it is possible to sort integers in time $O(n \cdot \log\log n)$, with no dependence on the key universe $[U]$, beating both MergeSort and RadixSort for large universe sizes (specifically, when $\log U = \omega(\log n \cdot \log\log n)$).

Overall, the Word-RAM Model has been the dominant model for analysis of algorithms for over 50 years, and thus has stood the test of time even as computing technology has evolved dramatically. It still provides a fairly accurate way of measuring how the efficiency of algorithms scales. That said, it is worth noting a few of the ways in which one can observe real-world performance that deviates from the Word-RAM model's predications:

1. Not all operations take the same amount of time. As discussed above, these differences typically lead to constant or logarithmic factors in runtime, which may be noticeable in practice.

2. The memory hierarchy. In real-life computers, not all memory accesses are equivalent. Reading from registers vs. cache vs. main memory vs. disk all take significantly different amounts

[2] Assuming that none of the input numbers are larger than n, we have $w = \lfloor \log S \rfloor + 1 = \lfloor \log_2(n+T) \rfloor + 1$ throughout the computation, where T is the amount of time taken. Initially, we have $S = n$ and there can be at most T MALLOC operations, so throughout we have $S \leq n + T$.

of time. It is possible to define computational models that account for the memory hierarchy, but they are beyond the scope of this course.

3. Parallelism. The RAM model captures only a single CPU operating sequentially, and doesn't model the performance we can gain from parallel computers (e.g. multi-core machines) or distributed computation. There are models for parallel and distributed algorithms, but also beyond the scope of this course.

Ignoring some of these aspects of computing technology is the price we pay for having a mathematically clean general-purpose theory of algorithms that lasts through changes in technology.

7 Word-RAM vs. RAM

Although the Word-RAM Model and the RAM Model each have their advantages (the Word-RAM is more realistic, while RAM is simpler), we can simulate each one by the other.

Theorem 7.1. *1. For every RAM program P, there is a Word-RAM Program P' such that P' halts on x iff P halts on x, and if they halt, then*[3] *$P'(x) = P(x)$ and*

$$\text{Time}_{P'}(x) = O\left((\text{Time}_P(x) + n + S) \cdot \left(\frac{\log M}{w_0}\right)^{O(1)}\right),$$

where n is the length of the input x, S is the largest memory location accessed by P on input x,[4] *M is the largest number computed by P on input x, and $w_0 = \lfloor \log \max\{n, x[0], \ldots, x[n-1]\} \rfloor + 1$.*

2. For every Word-RAM program P, there is a RAM program P' such that P' halts on x iff P halts on x, and if they halt, then $P'(x) = P(x)$ and

$$\text{Time}_{P'}(x) = O\left(\text{Time}_P(x) + n + w_0\right),$$

where n is the length of x and w_0 is the initial word size of P on input x.

Proof Sketch. 1.

2. Problem Set 3.

□

[3] Actually, if the result $P(x)$ does not fit into a single word, then P' will output a bignum representation of $P(x)$.

[4] Any RAM Program can be modified so that $S = O(n + \text{Time}_P(x))$ by using a Dictionary data structure, where the keys represent memory locations and the values represent numbers to be stored in those locations. The total number of elements to be stored in the data structure is bounded by n (the initial number of elements to be stored) and plus the number of writes that P performs, which is at most $\text{Time}_P(x)$. Depending on whether the Dictionary data structure is implemented using Balanced BSTs or Hash Tables, this transformation may incur a logarithmic-factor blow-up in runtime or yield a Las Vegas randomized algorithm.