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

Lecture 7: RAM and Word-RAM Simulations

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

1 Announcements

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.

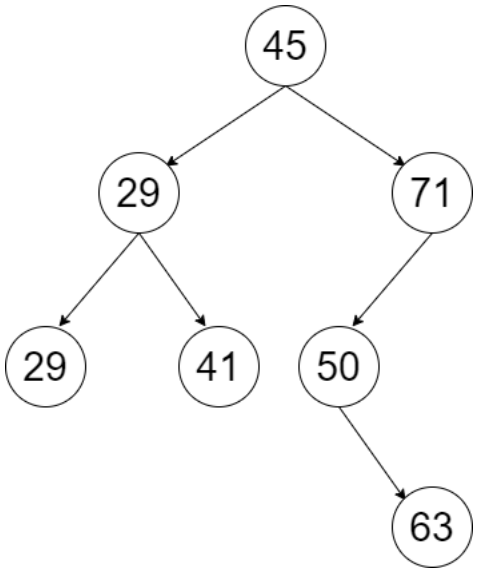
Theorem 2.2. *SortingOnNaturalNumbers can be done in time $O(n^2)$ in the RAM Model.*

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:

$$[45, 0, 4, 8, 29, 0, 12, 16, 71, 0, 29, 0, 20, 0, 0, 0, 41, 0, 0, 0, 50, 0, 0, 24, 63, 0, 0, 0]$$

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] 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

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"?

A:

Proof Idea. 1.

2.

□

Data encodings in simulations.

Efficiency of simulations.

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.

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 6.

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.

□

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.

2.

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.

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. □

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:

Using a finite word size leads us to the following computational model:

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

- Memory:
- Output:
- Operations:
- Variables:

- Crashing: A Word-RAM program *crashes* on input x and word length w if any of the following happen during its computation:

  1.
  2.

We denote the computation of a Word-RAM program on input x with word length w by $P[w](x)$. Note that $P[w](x)$ has one of three outcomes:

- halting with an output
- failing to halt, or
- crashing.

We define the *runtime* $T_{P[w]}(x)$ to be the number of commands executed until P either halts or crashes (so $T_{P[w]}(x) = \infty$ if $P[w](x)$ fails to halt).

This model, with say $w = 32$ or $w = 64$, is a reasonably good model for modern-day computers with 32-bit or 64-bit CPUs.

Q: What's wrong with just fixing $w = 64$ in Definition 6.1 and using it as our model of computation?

A:

Thus, we instead keep w as a varying parameter in Definition 6.1, which gives us a single program P that can be instantiated with different word lengths in order to handle arbitrarily large inputs (or computations that access arbitrarily large amounts of memory). We need to take care for how we define what it means for a Word-RAM program to solve a computational problem, and how we definite its runtime:

Definition 6.2. We say that word-RAM program P *solves* computational problem $\Pi = (\mathcal{I}, \mathcal{O}, f)$ if the following holds for every input x,

1. There is at least one word length $w \in \mathbb{N}$ such that $P[w](x)$ halts without crashing.
2. For every word length $w \in \mathbb{N}$ such that $P[w](x)$ halts without crashing, the output $P[w](x)$ satisfies $P[w](x) \in f(x)$ if $f(x) \neq \emptyset$ and $P[w](x) = \perp$ if $f(x) = \emptyset$.

Mental model: if $P[w](x)$ crashes, buy a better computer with a larger word size w and try again. Since larger word size generally means a more powerful computer, we expect the running time to decrease as the word-size gets larger, but we'll often want to run our program with whatever hardware we have in hand, or the smallest word size we can. Thus, it makes sense to measure complexity as follows:

Definition 6.3. The *running time* of a word-RAM program P on an input x is defined to be

$$T_P(x) =$$

Like in Lecture 2, we define the worst-case running time of P to be the function $T_P(n)$ that is the maximum of $T_P(x)$ over inputs x of size at most n.

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 justified by the following:

Proposition 6.4. *For a word-RAM program P and an input x that is an array of n numbers, if $T_P(x) < \infty$, then there is a word size w_0 such that $P[w](x)$ does not crash for any $w \geq w_0$. Specifically, we can take*

$$w_0 = \lceil \log_2 \max\{n + T_P(x), x[0], \ldots, x[n-1], c_0, \ldots, c_{k-1}\} + 1 \rceil,$$

where $c_0, \ldots, c_{k-1}$ are the constants appearing in variable assignments in P.

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.

Q: What's not captured by the Word-RAM model?

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' that simulates P. That is, for every input x, $T_{P'}(x) = \infty$ iff $T_P(x) = \infty$, and if both are finite, then for every word size w such that $P'[w](x)$ halts without crashing (which exists by Proposition 6.4), the output of $P'[w](x)$ is equal to (an encoding of) the output of $P(x)$. Furthermore,*

$$T_{P'[w]}(x) = O\left((T_P(x) + n + S) \cdot \left(\frac{\log M}{w}\right)^{O(1)}\right),$$

where n is the length of the input x, S is the largest memory location accessed by P on input x,[2] and M is the largest number computed by P on input x.

[2] Any RAM Program can be modified so that $S = O(n + T_P(x))$ 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 $T_P(x)$. 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.

2. *For every Word-RAM program P, there is a RAM program P' that simulates P in the sense that P' halts on (w, x) iff $P[w]$ halts on x, and if they halt, then the output of $P'(w, x)$ equals the output of $P[w](x)$ or* `crash` *according to whether $P[w](x)$ crashes or not. Furthermore,*

$$T_{P'}(w, x) = O\left(T_{P[w]}(x) + n + w\right),$$

where n is the length of x.

8 Takeaway

The Word-RAM model is the formal model of computation underlying everything we are doing in CS1200. Because of Proposition 6.4, we assume the word size is $w = O(\log(n + T(n)))$ for algorithms that run in time $T(n)$. After today, we will return to writing high-level pseudocode and won't torture ourselves with continuously writing Word-RAM code, but implicitly all of our theorems are about the Word-RAM programs that would be obtained by compiling our pseudocode into Word-RAM form. Thus, the Word-RAM model is the reference to use when we need to figure about how long some operation would take.

We 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 n, so can all be managed with a word length of $O(\log n)$. 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 the input size, then we need to pay closer attention and not treat arithmetic operations as constant time.