CS1200: Intro. to Algorithms and their Limitations Prof. Anurag Anshu

Lecture 6: RAM model, RAM simulations and Word-RAM model

Harvard SEAS - Spring 2026 *2026-02-18*

1 Announcements

- SRE2 next Mon.
- Anurag OH after class and Fri on zoom.
- Pset 3 released (due Feb 26).

2 Recommended Reading

- Hesterberg–Vadhan, Sections 5.1,5.2, 6.1-6.3, 7.1

3 Computational Models

In the last lecture, we mentioned various computational models, and stated their relevance in defining the complexity of an algorithm. What do we want from a computational model and a complexity measure?

- Unambiguity.
- Expressivity.
- Mathematical simplicity.
- Robustness.
- Technological relevance.

4 The RAM Model

Our first attempt at a precise model of computation is the *RAM model*, which models memory as an infinite array M of *natural numbers*.

Definition 4.1 (RAM Programs: syntax)**.** A *RAM Program* $P = (V, C_0, \ldots, C_{\ell-1})$ consists of a finite set V of *variables* (or *registers*), and a sequence $C_0, C_1, \ldots, C_{\ell-1}$ of *commands* (or *lines of code*), each chosen from the following:

- (assignment to a constant)
- (arithmetic)
- (read from memory)

- (write to memory)
- (conditional goto)

In addition, we require that every RAM Program has three special variables:

Definition 4.2 (Computation of a RAM Program: semantics)**.** A RAM Program $P = (V, (C_0, \ldots, C_{\ell-1}))$ *computes* on an input x as follows:

1. Initialization:

2. Execution:

3. Output:

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

The definition of the RAM Model above is mathematically precise, so it achieves our unambiguity desideratum (unless we've forgotten to specify something!).

The RAM Model also does quite well on the mathematical simplicity front. We described it in one page of text in this book, compared to 100+ pages for most modern programming languages. That said, there are even simpler models of computation, such as Turing Machines and the Lambda Calculus. However, those are harder to describe algorithms in and less accurately describe computing technology. We will briefly discuss those later in the course when we cover the Church–Turing Thesis, and they (along with other models of computation) are studied in depth in CS1210.

5 Iterative Algorithms

Theorem 5.1. *Algorithm 5.1 solves* Sorting On Natural Numbers *in time* $O(n^2)$ *in the RAM Model.*

We present the low-level RAM code to convince you that this can be done in principle, but outside that context one does not generally read or write low-level RAM code (because that would

be tedious).

```
   InsertionSort(x):
   Input      : An array x = (K_0, V_0, K_1, V_1, ..., K_{n-1}, V_{n-1}), occupying memory
                locations M[0], ..., M[2n-1]
   Output     : A valid sorting of x in the same memory locations as the input
   Variables  : input_len, output_len, zero, one, two, output_ptr, outer_key_ptr,
                outer_rem, outer_key, inner_key_ptr, inner_rem, inner_key, key_diff,
                insert_key, insert_value, temp_ptr, temp_key, temp_value
 0 zero = 0 ;                                                  /* useful constants */
 1 one = 1;
 2 two = 2;
 3 output_ptr = 0 ;                                  /* output will overwrite input */
 4 output_len = input_len + zero;
 5 outer_key_ptr = 0 ;                    /* pointer to the key we want to insert */
 6 outer_rem = input_len/two ;              /* # outer-loop iterations remaining */
 7    outer_key_ptr = outer_key_ptr + two ;                /* start of outer loop */
 8    outer_rem = outer_rem - one;
 9    IF outer_rem == 0 GOTO 34;
10    outer_key = M[outer_key_ptr] ;                        /* key to be inserted */
11    inner_key_ptr = 0 ;                 /* pointer to potential insertion point */
12    inner_rem = outer_key_ptr/two ;       /* # inner-loop iterations remaining */
13       inner_key = M[inner_key_ptr] ;                  /* start 1st inner loop */
14       key_diff = inner_key - outer_key ;     /* if inner_key ≤ outer_key, then */
15       IF key_diff == 0 GOTO 30 ;          /* proceed to next inner iteration */
16       insert_key = outer_key + zero ;                  /* key to be inserted */
17       temp_ptr = outer_key_ptr + one;
18       insert_value = M[temp_ptr] ;                    /* value to be inserted */
19          temp_key = M[inner_key_ptr] ;             /* start of 2nd inner loop */
20          temp_ptr = inner_key_ptr + one;
21          temp_value = M[temp_ptr];
22          M[inner_key_ptr] = insert_key;
23          M[temp_ptr] = insert_value;
24          insert_key = temp_key + zero;
25          insert_value = temp_value + zero;
26          inner_key_ptr = inner_key_ptr + two;
27          IF inner_rem == 0 GOTO 7;
28          inner_rem = inner_rem - one;
29          IF zero == 0 GOTO 19;
30       inner_key_ptr = inner_key_ptr + two;
31       inner_rem = inner_rem - one;
32       IF inner_rem == 0 GOTO 7;
33       IF zero == 0 GOTO 13;
34 HALT ;                                                 /* not an actual command */
```

Algorithm 5.1: RAM implementation of `InsertionSort()`

6 Expressiveness: Simulating High-Level Programs

So far, 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.

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 6.1 (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"?

A:

Proof Idea. 1.

2.

□

[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 **??**.

Data encodings in simulations.

Efficiency of simulations.

7 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 7.1. *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.*

Corollary 7.2. *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.

8 Technological Relevance

In Theorem 6.1, 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 next.

We address Issue 1 via another simulation theorem:

Theorem 8.1. *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 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 $\texttt{output\_len}'$ 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. □

9 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 9.1. The *Word-RAM program* P is defined like a RAM program except that it has a (static) *word length* parameter w and a (dynamic) *memory size* S. These are used as follows:

- Memory:
- Output:
- Operations:
- Crashing: A Word-RAM program P *crashes* on input x and word length w if any of the following occur:
  1.
  2.
  3.

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). Two sample Word-RAM Programs are given as Algorithms 2 and 3.

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 9.1 and using it as our model of computation?

```
Duplicate(x):
Input      : An array x of length n
Output     : The array x ∘ x, i.e. x concatenated with itself.
Variables  : input_len, output_ptr, output_len, input_ctr, output_ctr,
             temp, zero, one, three
0  zero = 0;
1  one = 1;
2  three = 3;
3  input_ctr = 0;
4  output_ctr = input_len;
5     temp = input_len − input_ctr;
6     IF temp == 0 GOTO 13;
7     temp = M[input_ctr];
8     MALLOC;
9     M[output_ctr] = temp;
10    input_ctr = input_ctr + one;
11    output_ctr = output_ctr + one;
12    IF zero == 0 GOTO 5;
13 output_ptr = 0;
14 output_len = three × input_len;
```

Algorithm 9.1: A Word-RAM program to illustrate the use of MALLOC and the output truncation. At the end of the program, $S = 2n$ but $\texttt{output\_len} = 3n$, so the output is truncated to end at $M[S-1]$.

A:

Since Definition 9.1 allows the same word-RAM program P to be instantiated with different word lengths, 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 define its runtime:

Definition 9.2. We say that a word-RAM program P *solves* a computational problem $\Pi = (\mathcal{I}, \mathcal{O}, f)$ if the following hold for every input $x \in \mathcal{I}$:

1.

2.

3.

Mental model: if $P[w](x)$ crashes, buy a better computer with a larger word size w and try again. With this definition, it can be verified that Algorithms 2 and 3 solve the computational problems they claim to solve.

Definition 9.3. The *running time* of a word-RAM program P on an input x is defined to be

$$T_P(x) =$$

```
   SameSum(x):
   Input      : An array x of length 4
   Output     : 1 if x[0] + x[1] = x[2] + x[3], 0 otherwise
 0 base = 2;
 1     nextbase = base × 2;
 2     diff = nextbase + 1;
 3     diff = diff − nextbase;
 4     IF diff == 0 GOTO 7;
 5     base = nextbase / 2;
 6     GOTO 1;
 7 FOR i = 0 to 3 DO;
 8     MALLOC; MALLOC;
 9     M[4 + 2 ∗ i] = M[i]%base;
10     M[5 + 2 ∗ i] = M[i]/base;
11 FOR i = 0 to 3 DO;
12 prefixlowsum = M[4] + M[6];
13 prefixhighsum = M[5] + M[7];
14 prefixsumlow = prefixlowsum%base;
15 prefixsumhigh = prefixlowsum/base + prefixhighsum;
16 Similarly calculate suffixsumlow, suffixsumhigh from M[8] + M[10] and M[9] + M[11];
17 IF prefixsumlow == suffixsumlow AND prefixsumhigh == suffixsumhigh THEN
     M[0] = 1;
18 ELSE M[0] = 0;
19 output_ptr = 0; output_len = 1;
```

Algorithm 9.2: A Word-RAM program (with a lot of shorthand for readability) to illustrate the use of bignum and saturation arithmetic. If the program doesn't crash, we know that $x[i] < 2^w$ for each i, but the sum of two elements of $x[i]$ can exceed 2^w. Thus, we calculate $\texttt{base} = 2^{w-1}$ as the largest power of two such that $\texttt{base}+1$ does not saturate at $2^w - 1$, and then decompose each $x[i]$ into its base base representation (with the units digit in $M[4+2i]$ and the $\texttt{base} = 2^{w-1}$ digit in $M[4+2i+1]$). We then calculate the sums by the grade-school addition method, noting that there is no carry past the second digit since the sum of any two input numbers is smaller than $2 \cdot 2^w \leq 2^{2w-2} = \texttt{base}^2$, since $w \geq 3$ due to the presence of the constant 4 in the program. (If we needed to do bignum multiplication, it would be better to have $\texttt{base} = 2^{w/2}$, so that the product of two digits fits always within a word.) This program uses some operations that are not part of the Word-RAM syntax, but each of them can be decomposed into Word-RAM operations. Since the FOR loop has a constant number of iterations, it can be unrolled. The mod (%) operation can be simulated as in Theorem 7.1.

As we have been doing throughout the course, 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.

Q: What are the running times of Algorithm 2 and 3?

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 9.4. *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 aspects of "computational efficiency in practice" not captured by the Word-RAM model?