

## Lecture 6: The RAM Model

*Harvard SEAS - Fall 2024**2024-09-19*

## 1 Announcements

Recommended Reading:

- CLRS Sec 2.2
- From syllabus: “All sources of ideas, including the names of any collaborators (including AI tools, if used for brainstorming), must be listed on your submitted homework along with a brief description of how they influenced your work. Github Copilot and similar tools should be turned off when working on programming assignments.” Starting with ps2, there is a place to do so.
- Fill out participation/reflection survey after SRE.
- OH use: not for grading. Remember revision policy, and up to 4 R- grades on psets is still in A-range.
- Post questions for course staff on Ed (including about SRE logistics, excused absences, etc.), not by email.
- Let us know about excused absences from SREs prior to the previous lecture; you will do the SRE with the DCE students.
- PS0 feedback
  - Median time = 7hrs, 75th percentile = 9hrs. Much better than past years (e.g. 2022 had median = 10hrs, 75th percentile = 15hrs).
  - Most students spent most time on figuring out how to solve the theory problems. That’s good!
  - “The induction proof felt a little counter-intuitive and difficult to do without office hours, but it was great otherwise! I wish we had more coding questions, though!”
  - “I overheard lots of people who were too intimidated to start early because they were unfamiliar with github and overleaf”
  - 75%+ feel that background for ps0 was covered in prior courses, both for theory background and for programming background.
  - Majority consider lecture pacing “just right,” rest are split between too fast and too slow.
  - “I believe that when we start a proof in lectures, especially if it’s a more complex one, we should aim to complete it in full. For example, I felt like we moved too quickly through the merge sort proof, and it was left to us to review it in the notes rather than going through the entire process together in class.”

- “Often feels as if the class expects knowledge we barely covered in past classes- ie. what is omega and how do we actually prove this.”
- “It would be amazing if the TFs could cover the whole section notes – in my section the TFs did not elaborate on the asymptotic notation part, which I feel like I need more practice on.”
- “I wish that if everyone was working on the same problem, we could do a group help session rather than helping everyone individually via a queue. Otherwise - Office Hours were very helpful.”
- “A queue would be very nice”

## 2 Computational Models

So far, our conception of an algorithm has been informal: “a well-defined procedure for transforming inputs to outputs” whose runtime is measured as “basic operations” performed on a given input. This is unsatisfactory: how can we identify the fastest algorithm to solve a given problem if we don’t have agreement on what counts as an algorithm or as a basic operation?

To address this, we need to specify a *computational model*. A computational model is any precise way of describing computations. It is a broad term, which encompasses abstract mathematical models, programming languages, and models of computer hardware. They can be organized according to how close or far they are to describing how computations should be physically executed. In approximate order from “far from physical” to “close to physical,” some examples of computational models are:

0. (Natural languages: English, ChatGPT)
1. Declarative and functional programming languages: OCaml, Lisp, Prolog, SQL, the Lambda Calculus
2. Imperative high-level programming languages: Python, R
3. “Close to the metal” imperative programming languages: C, Rust
4. Architecture-level models: Assembly language, (Word-)RAM Model, Turing machines, Quantum Turing machines
5. Hardware models: Boolean circuits, Quantum circuits, (Bio-inspired) Cellular automata, DNA computers

Of course, in the end, even programs in the highest level of this hierarchy (like OCaml programs) can be executed on the same hardware that is modelled by the lowest level of this hierarchy (e.g. Boolean circuits). This translation is done by *compilers* or *interpreters* that translate between different models of computation.

Since we are interested in the efficiency of algorithms, we also need to associate with our computational model one or more complexity measures that model how much “time” an algorithm takes on different inputs (or other resources, like space or energy usage). This is easiest for the lowest levels in the hierarchy, which offer small units of “basic operations” that we can measure.

But as we will see, even this has some subtlety; sometimes operations that seem basic can have widely varying physical runtimes when compiled and run on hardware in practice.

What do we want from a computational model and a complexity measure?

- **Unambiguity.** Precisely defines what is and isn't an algorithm (both the syntax and semantics), and how we measure its runtime (e.g. exactly what counts as a “basic operation”). English and ChatGPT do not meet this criterion, so do not qualify as computational models.
- **Expressivity.** Everything we intuitively consider to be an algorithm should be expressible, or we should have a good reason for excluding it.
- **Mathematical simplicity.** We want something that is manageable to reason about. In contrast, the specification of most modern programming languages is too long and cumbersome to prove anything general by hand.<sup>1</sup> For example, the Python Language Reference (Release 3.4.3), which “describes the syntax and ‘core semantics’ of the language” (not including libraries) is 135 pages long.
- **Robustness.** Small changes to the model should not fundamentally change what can or cannot be computed, and should have only a small effect on efficiency.
- **Technological relevance.** The model and its complexity measure should capture reasonably well the implementability and efficiency of algorithms on actual computing hardware, even as technology evolves with time.

### 3 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 3.1** (RAM Programs: syntax). A *RAM Program*  $P = (V, C_0, \dots, C_{\ell-1})$  consists of a finite set  $V$  of *variables* (or *registers*), and a sequence  $C_0, C_1, \dots, C_{\ell-1}$  of *commands* (or *lines of code*), each chosen from the following:

- (assignment to a constant)  $\text{var} = c$ , for a variable  $\text{var} \in V$  and a constant  $c \in \mathbb{N}$ .
- (arithmetic)  $\text{var}_0 = \text{var}_1 \text{ op } \text{var}_2$ , for variables  $\text{var}_0, \text{var}_1, \text{var}_2 \in V$ , and an operation  $\text{op}$  chosen from  $+, -, \times, /$ .
- (read from memory)  $\text{var}_0 = M[\text{var}_1]$  for variables  $\text{var}_0, \text{var}_1 \in V$ .
- (write to memory)  $M[\text{var}_0] = \text{var}_1$  for variables  $\text{var}_0, \text{var}_1 \in V$ .
- (conditional goto) IF  $\text{var} == 0$ , GOTO  $k$ , where  $k \in \{0, 1, \dots, \ell\}$ .<sup>2</sup>

In addition, we require that every RAM Program has three special variables: `input_len`, `output_ptr`, and `output_len`.

---

<sup>1</sup>As we'll discuss at the end of the course, there are software tools that assist in proving properties of programs. However, to make effective use of these tools you need to have experience in proving things about programs yourself!

<sup>2</sup>The intended semantics of this are that if the current value stored in `var` is 0, then the program execution should jump to line number  $k$ .

**Definition 3.2** (Computation of a RAM Program: semantics). A RAM Program  $P = (V, (C_0, \dots, C_{\ell-1}))$  *computes* on an input  $x$  is as follows:

1. Initialization: The input  $x$  is encoded (in some predefined manner) as a sequence of natural numbers placed into memory locations  $(M[0], \dots, M[n-1])$ , and all of the remaining memory locations are set to 0. The variable `input_len` is initialized to  $n$ , the length of  $x$ 's encoding. All other variables are initialized to 0.
2. Execution: The sequence of commands  $C_0, C_1, C_2, \dots$  are executed in order (except when jumps are done due to GOTO commands), updating the values of variable and memory locations according to the usual interpretations of the operations. Since we are working with natural numbers, if the result of subtraction would be negative, it is replaced with 0. Similarly, the results of division are rounded down, and divide by 0 results in 0.
3. Output: If line  $\ell$  is reached (possibly due to a GOTO  $\ell$ ), the output  $P(x)$  is defined to be the subarray of  $M$  of length `output_len` starting at location `output_ptr`. That is,

$$P(x) = (M[\text{output\_ptr}], M[\text{output\_ptr} + 1], \dots, M[\text{output\_ptr} + \text{output\_len} - 1]).$$

The *running time* of  $P$  on input  $x$ , denoted  $T_P(x)$ , is defined to be: the number of commands executed during the computation (possibly  $\infty$ ).

The definition of the RAM Model above is mathematically precise, so 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 lecture notes, compared to 100+ pages for most modern programming languages. That said, there are even simpler models of computation, such as the Turing Machine 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.

Our focus for the rest of today's class will be to get convinced of the *expressivity* of the RAM model. We will do this by seeing how to implement algorithms we have seen in the RAM model. We will turn to robustness and technological relevance next time.

## 4 Iterative Algorithms

Let's see an example RAM program for Insertion Sort. This is just a low-level implementation of the same algorithm that we saw in Lecture 1. Since RAM programs can only take inputs that are arrays of natural numbers, this program solves the corresponding restriction of the Sorting problem, which we call `SortingOnNaturalNumbers`. We can check that our runtime analysis from Lecture 2 applies to this program; that is, everything we counted as a basic operation there takes only  $O(1)$  RAM program steps. Thus, we have:

**Theorem 4.1.** *Algorithm 1 solves `SortingOnNaturalNumbers` in time  $O(n^2)$  in the RAM Model.*

Our goal in the rest of this lecture is to get convinced that all of the algorithm and data structure analyses we have done so far in the course can analogously be made completely precise and rigorous in the RAM Model. We are presenting the low-level RAM code to convince you that this can be

done in principle, but once we are past this point, you won't be expected to read or write low-level

RAM code for the algorithms we are studying class (because that would be tedious and painful).

<b>Input</b>	: An array $x = (K_0, V_0, K_1, V_1, \dots, K_{n-1}, V_{n-1})$ , occupying memory locations $M[0], \dots, M[2n - 1]$
<b>Output</b>	: A valid sorting of $x$ . in the same memory locations as the input
<b>Variables</b>	: 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 1:** RAM implementation of Insertion Sort

## 5 Data

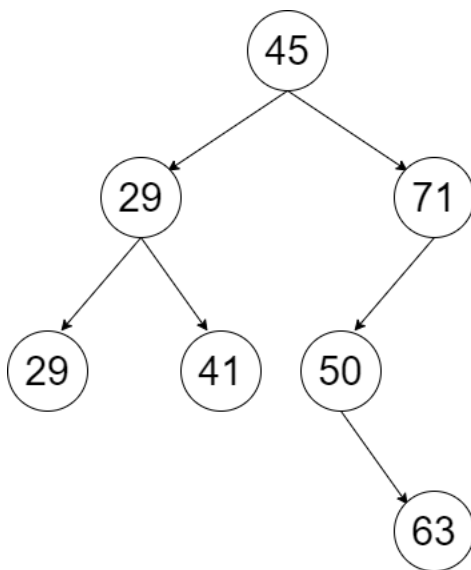
Implicit in the expressivity requirement is that we can describe the inputs and outputs of algorithms in the model. In the RAM model, all inputs and outputs are arrays of natural numbers. How can we represent other types of data?

- (Signed) integers: natural numbers with an additional sign bit
- Rational numbers: pair of integers (numerator and denominator)
- Real numbers: impossible! (For those of you who have taken CS20, this is because the set of real numbers is uncountable, but the set of finite sequences of natural numbers is countable.) Although this is a constraint on expressivity, the inability to represent and manipulate infinite-precision real numbers is generally accepted as part of both the intuitive and physically realizable concepts of computation.<sup>3</sup>

To approximate real numbers, Python and other programming languages often use *floating-point numbers*, in the form  $x \cdot 2^y$  for signed integers  $x$  and  $y$  of bounded bit-length. Almost every operation on floating-point numbers introduces error through rounding, and it can be very tricky to reason about how these errors propagate. Thus, *in the rest of the course, we will generally stick to problems involving integer or rational arithmetic*. Dealing with floating-point errors is a topic sometimes covered in courses on numerical analysis, scientific computation, numerical linear algebra, and/or optimization, which you might find in the Applied Math or Applied Computation parts of the catalogue.

- Strings: an array of ASCII values (where each character is represented as a number in  $\{0, 1, \dots, 127\}$ )

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



---

<sup>3</sup>That said, one can define real-number models of computation, and these are mathematically interesting to study. See the book *Complexity and Real Computation* by Blum, Shub, and Smale.

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.

## 6 Recursive Algorithms

*We will not cover this in lecture, but include it here in case you are interested and/or want more convincing about the expressivity of the RAM Model.*

It is not entirely obvious that the RAM Model can implement recursion, since there are no function calls in its description. The way this is done (both in theory and in practice) via the use of a *stack* data structure. We simulate a function call  $f(x)$  through the following steps:

1. Push local variables (in scope of the calling code), the input  $x$ , and an indicator of which line number to return to after the  $f$  is done executing.
2. GOTO the line number that starts the implementation of  $f$ .
3. The implementation of  $f$  pops its input  $x$  off the top of the stack, computes  $y = f(x)$ , and pushes  $y$  onto the top of the stack, and then GOTO to the line number after the function call (which it also read off the top of the stack).
4. After the return,  $y = f(x)$  is read off the top of the stack, along with the local variables needed to continue the computation where it left off before calling  $f$ .

Below we present an example for a recursive computation of the height of a binary tree. Since our RAM model doesn't allow negative numbers, our recursive functions will compute height plus one (so that an empty tree has height 0), and we will subtract one from the height at the end. Also, because this algorithm does not use any memory other than the stack and the arrays, we implement the stack as a contiguous segment of memory starting after the input. However, in general, one may need to implement it as a linked list in order to be able to skip over portions of memory that



are being used for global state.

<b>Input</b>	: A Binary Tree of Key-Value Pairs, given as an array of 4-tuples ( $K, V, P_L, P_R$ )
<b>Output</b>	: The height of the input tree
0	zero = 0 ; <span style="float: right;">/* useful constants */</span>
1	one = 1;
2	two = 2;
3	stack_ptr = input_len + zero;
4	M[stack_ptr] = zero ; <span style="float: right;">/* push pointer to root of tree to top of stack */</span>
5	stack_ptr = stack_ptr + one;
6	M[stack_ptr] = zero ; <span style="float: right;">/* branch-indicator for root call */</span>
7	branch = M[stack_ptr] ; <span style="float: right;">/* pop branch indicator */</span>
8	stack_ptr = stack_ptr - one;
9	node_ptr = M[stack_ptr] ; <span style="float: right;">/* pop pointer to current node */</span>
10	stack_ptr = stack_ptr + two ; <span style="float: right;">/* and repush both back onto stack */</span>
11	temp_ptr = node_ptr + two;
12	child_ptr = M[temp_ptr] ; <span style="float: right;">/* pointer to left child */</span>
13	IF child_ptr == 0 GOTO 22;
14	M[stack_ptr] = child_ptr ; <span style="float: right;">/* push pointer to left child */</span>
15	stack_ptr = stack_ptr + one;
16	M[stack_ptr] = one ; <span style="float: right;">/* left branch indicator */</span>
17	IF zero == 0 GOTO 7 ; <span style="float: right;">/* recurse */</span>

**Algorithm 2:** RAM implementation of Calculate Height

```

18  left_height = M[stack_ptr] ;           /* pop height of left child */
19  stack_ptr = stack_ptr - one;
20  node_ptr = M[stack_ptr] ;             /* pop pointer to current node */
21  IF zero == 0 GOTO 23
22  left_height = 0 ;                     /* left child is empty */
23  temp_ptr = node_ptr + three;
24  child_ptr = M[temp_ptr] ;             /* pointer to right child */
25  IF child_ptr == 0 GOTO 37;
26  M[stack_ptr] = left_height ;          /* push height of left child */
27  stack_ptr = stack_ptr + one;
28  M[stack_ptr] = child_ptr ;            /* push pointer to right child */
29  stack_ptr = stack_ptr + one;
30  M[stack_ptr] = two ;                  /* right branch indicator */
31  IF zero == 0 GOTO 7 ;                  /* recurse */
32  right_height = M[stack_ptr] ;          /* pop height of right child */
33  stack_ptr = stack_ptr - one;
34  left_height = M[stack_ptr] ;           /* pop height of left child */
35  stack_ptr = stack_ptr - one;
36  IF zero == 0 GOTO 38;
37  right_height = 0
38  branch = M[stack_ptr] ;               /* pop branch indicator */
39  diff_heights = left_height - right_height;
40  IF diff_heights == 0 GOTO 43 ;         /* right-child taller */
41  height = left_height + one ;          /* left-child taller */
42  IF zero == 0 GOTO 44;
43  height = right_height + one;
44  IF branch == zero GOTO 50;
45  M[stack_ptr] = height ;               /* push return value */
46  branch = branch - one;
47  IF branch == zero GOTO 18;
48  branch = branch - one;
49  IF branch == zero GOTO 32;
50 height = height - one ;               /* subtract one for output height */
51 M[stack_ptr] = height;
52 output_ptr = stack_ptr;
53 output_len = 1;
54 HALT

```

**Algorithm 3:** RAM implementation of Calculate Height (cont.)

## 7 Reductions

*We may not have time to cover this in class, but is again included for your interest.*

We can also formalize *reductions* using the following extension of the RAM model.

**Definition 7.1.** An *oracle-aided RAM Program* is like an ordinary RAM program, except it can also have commands of the form

ORACLE( $\text{var}_0, \text{var}_1, \text{var}_2$ ),

which means call the oracle on the array ( $M[\text{var}_0], M[\text{var}_0 + 1], \dots, M[\text{var}_0 + \text{var}_1 - 1]$ ) and write the oracle's answer in the locations ( $M[\text{var}_2], M[\text{var}_2 + 1], \dots$ ).

For example, our reduction from IntervalScheduling-Decision to Sorting is given by the following oracle-aided RAM program:

<b>Input</b>	: An array $x = (a_0, b_0, a_1, b_1, \dots, a_{n-1}, b_{n-1})$ , occupying memory locations $M[0], \dots, M[2n - 1]$ , with $a_i \leq b_i$ for all $i$		
<b>Output</b>	: 1 (YES) if all of the intervals $[a_i, b_i]$ are disjoint, 0 (NO) otherwise		

```

0 zero = 0;
1 one = 1;
2 two = 2;
3 ORACLE(zero, input_len, zero) ;           /* sort input by start time */
4 output_ptr = 0;
5 output_len = 1;
6 M[zero] = one ;                           /* default output is 1 = YES */
7 temp_ptr = 1;
8 remaining = input_len - two ; /* how many adjacent pairs left to check, times
   two */
9   IF remaining == 0 GOTO 19;
10  end_curr = M[temp_ptr] ;                  /* read end time of current interval */
11  temp_ptr = temp_ptr + one;
12  start_next = M[temp_ptr] ;                /* read start time of next interval */
13  temp = start_next - end_curr;
14  IF temp == 0 GOTO 18 ;                    /* conflict found */
15  temp_ptr = temp_ptr + one;
16  remaining = remaining - two;
17  IF zero == 0 GOTO 9;
18 M[zero] = zero ;                          /* change output to 0 = NO */
19 HALT

```

**Algorithm 4:** Oracle-RAM implementation of IntervalScheduling-Decision  $\leq_{O(n),n}$  Sorting