CS120: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Lecture 7: The RAM Model

Harvard SEAS - Fall 2021 *Sept. 23, 2021*

1 Announcements

Recommended Reading:

- CLRS Sec 2.2
- Salil OH 4-5 today 3.327
- PS3 Posted!
- Revisions for PS0+PS1 due end of this weekend
- Remember to bring name placards

2 The RAM Model

So far, we've been informal in our treatment of algorithms and their runtime. To turn everything into actual mathematical theorems, we need to have a precise mathematical model of computation.

The RAM Model: memory is an infinite array M of *natural numbers*.

Definition 2.1 (RAM Programs)**.** A *RAM Program* $P = (V, C_0, \ldots, C_{\ell-1})$ consists of a finite set V of *variables*, and a sequence $C_0, C_1, \ldots, C_{\ell-1}$ of commands, chosen from the following:

- (assignment to a constant) $\texttt{var} = c$, for a variable $\texttt{var} \in V$ and a constant $c \in \mathbb{Z}$.
- (arithmetic) $\texttt{var}_0 = \texttt{var}_1$ `op` $\texttt{var}_2$, for variables $\texttt{var}_0, \texttt{var}_1, \texttt{var}_2 \in V$, and an operation `op` chosen from $+$, $-$, $\times$, $/$. (Since we are working with natural numbers, if the result of subtraction would be negative, it is replaced with 0. And the results of division are rounded down.)
- (read from memory) $\texttt{var}_0 = M[\texttt{var}_1]$ for variables $\texttt{var}_0, \texttt{var}_1 \in V$.
- (write to memory) $M[\texttt{var}_0] = \texttt{var}_1$ for variables $\texttt{var}_0, \texttt{var}_1 \in V$.
- (conditional goto) IF $\texttt{var} == 0$, GOTO k, where $k \in \{0, 1, \ldots, \ell - 1\}$.
- (halt) HALT.

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

Definition 2.2 (Computation of a RAM Program)**.** A RAM Program $P = (V, (C_0, \ldots, C_{\ell-1}))$ *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 $(M[0], \ldots, M[n-1])$, 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 $C_0, C_1, C_2, \ldots$ are executed in order (except when jumps are done due to GOTO commands), updating the values of variable and memory locations according to the descriptions of the operations above.

3. Output: When a HALT command or line ℓ is reached, the output $P(x)$ is defined to be the subarray of M of length `output_len` starting at location `output_ptr`. That is,

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

   If P does not halt on x, then we define $P(x) = \perp$.

The *running time* of P on input x is defined to be: the number of commands executed during the computation (possibly ∞).

Let's see an example RAM program for Insertion Sort, when the items and keys are both given as natural numbers.

Input : An array $A = ((I_0, K_0), \ldots, (I_{n-1}, K_{n-1}))$, occupying memory locations $M[0], \ldots, M[2n-1]$

Output : A valid sorting of A. in the same memory locations as the input

Variables: input_len, output_ptr, output_len, zero, one, two, outer_key_ptr, outer_rem, inner_key_ptr, inner_rem, inner_key, key_diff, insert_key, outer_key, temp_ptr, temp_key, temp_item

```
1  zero = 0 ;                                              /* useful constants */
2  one = 1;
3  two = 2;
4  output_ptr = zero ;                                  /* output will overwrite input */
5  output_len = input_len + zero;
6  outer_key_ptr = 1 ;                          /* pointer to the key we want to insert */
7  outer_rem = input_len/two ;                     /* # outer-loop iterations remaining */
8     outer_key_ptr = outer_key_ptr + two ;                  /* start of outer loop */
9     outer_rem = outer_rem − one;
10    IF outer_rem == 0 GOTO 35;
11    outer_key = M[outer_key_ptr] ;                          /* key to be inserted */
12    inner_key_ptr = 1 ;                          /* pointer to potential insertion point */
13    inner_rem = outer_key_ptr/two ;              /* # inner-loop iterations remaining */
14       inner_key = M[inner_key_ptr] ;                     /* start 1st inner loop */
15       key_diff = inner_key − outer_key ;     /* if inner_key ≤ outer_key, then */
16       IF key_diff == 0 GOTO 31 ;          /* proceed to next inner iteration */
17       insert_key = outer_key + zero ;                     /* key to be inserted */
18       temp_ptr = outer_key_ptr − one;
19       insert_item = M[temp_ptr] ;                        /* item to be inserted */
20          temp_key = M[inner_key_ptr] ;                /* start of 2nd inner loop */
21          temp_ptr = inner_key_ptr − 1;
22          temp_item = M[temp_ptr];
23          M[temp_ptr] = insert_item;
24          M[inner_key_ptr] = insert_key;
25          insert_key = temp_key + zero;
26          insert_item = temp_item + zero;
27          inner_key_ptr = inner_key_ptr + two;
28          inner_rem = inner_rem − one;
29          IF inner_rem == 0 GOTO 8;
30          IF zero == 0 GOTO 20;
31       inner_key_ptr = inner_key_ptr + two;
32       inner_rem = inner_rem − one;
33       IF inner_rem == 0 GOTO 8;
34       IF zero == 0 GOTO 14;
35 HALT
```

Algorithm 1: RAM implementation of Insertion Sort

Theorem 2.3. *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.

Proof idea:

1. Compilers! Python programs (or rather their bytecode) can be emulated by an interpreter written in C (or Java) that is compiled to x86 assembly language, which very similar to (and easily simulated by) our model of RAM programs. (In assembly language, what we are calling *variables* are referred to as *registers*, and these represent physical storage locations in the CPU. For this reason, the RAM Model also usually has the terminology "registers".)

2. 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. (See Problem Set 4.)

□

Q: Isn't assembly language more powerful than our RAM model?

Those of you familiar with assembly language allows a wider set of operations that we've allowed here, and indeed people define RAM models with varying sets of operations. One reason we use asymptotic notation like $O(\cdot)$ is so that these details don't really matter. We establish robustness of our model by *simulation theorems* like the following:

Theorem 2.4. *Let P be a program in an extended RAM Model, where we also allow a mod (%) operation. Then there is a standard RAM program P' such that for every input x, $P'(x) = P(x)$ and the runtime of P' 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 $\texttt{var}_t$, and replace this line of code with:
$\texttt{var}_t = \texttt{var}_1 / \texttt{var}_2$
$\texttt{var}_t = \texttt{var}_t * \texttt{var}_2$
$\texttt{var}_0 = \texttt{var}_1 - \texttt{var}_t$

□

Given Theorem 2.3 and simulation results like Theorem 2.4, we can define the (asymptotic) runtime of a Python program (or any algorithm we describe) as the (asymptotic) runtime of its RAM implementation.

You might also be wondering:
Q: Isn't the RAM Model more powerful than assembly language?

For example:

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).

You will address Issue 1 in the next active-learning exercise. Issue 2 will be addressed next time, when we introduce the Word-RAM model.

We'll come back to this in Section 4 and in an exercise (either active learning or on the problem set).

3 Recursion on a RAM

[Not discussed in detail in lecture, and included here only for "culture" and/or in case you want more convincing about Theorem 2.4.]

One thing that may be particularly nonobvious about Theorem 2.4 is how 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.

Input : A Binary Tree of Item-Key Pairs, given as an array of 4-tuples (I, K, P_L, P_R)
Output : The height of the input tree
Variables:

```
1  zero = 0 ;                                              /* useful constants */
2  one = 1;
3  two = 2;
4  stack_ptr = input_len + zero;
5  M[stack_ptr] = zero ;        /* push pointer to root of tree to top of stack */
6  stack_ptr = stack_ptr + one;
7  M[stack_ptr] = zero ;                    /* branch-indicator for root call */
8     branch = M[stack_ptr] ;                          /* pop branch indicator */
9     stack_ptr = stack_ptr − one;
10    node_ptr = M[stack_ptr] ;                  /* pop pointer to current node */
11    stack_ptr = stack_ptr + two ;          /* and repush both back onto stack */
12    child_ptr = node_ptr + two ;                    /* pointer to left child */
13    IF child_ptr == 0 GOTO 22;
14    M[stack_ptr] = child_ptr ;                /* push pointer to left child */
15    stack_ptr = stack_ptr + one;
16    M[stack_ptr] = one ;                          /* left branch indicator */
17    IF zero == 0 GOTO 8 ;                                        /* recurse */
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    child_ptr = node_ptr + three ;               /* pointer to right child */
24    IF child_ptr == 0 GOTO 36;
25    M[stack_ptr] = left_height ;              /* push height of left child */
26    stack_ptr = stack_ptr + one;
27    M[stack_ptr] = child_ptr ;                /* push pointer to right child */
28    stack_ptr = stack_ptr + one;
29    M[stack_ptr] = two ;                          /* right branch indicator */
30    IF zero == 0 GOTO 8 ;                                        /* recurse */
31    right_height = M[stack_ptr] ;             /* pop height of right child */
32    stack_ptr = stack_ptr − one;
33    left_height = M[stack_ptr] ;               /* pop height of left child */
34    stack_ptr = stack_ptr − one;
35    IF zero == 0 GOTO 37;
36    right_height = 0
```

Algorithm 2: RAM implementation of Calculate Height

```
37    branch = M[stack_ptr] ;                          /* pop branch indicator */
38    diff_heights = left_height − right_height;
39    IF diff_heights == 0 GOTO 42 ;                    /* right-child taller */
40    height = left_height + one ;                      /* left-child taller */
41    IF zero == 0 GOTO 43;
42    height = right_height + one;
43    IF branch == zero GOTO 49;
44    M[stack_ptr] = height ;                           /* push return value */
45    branch = branch − one;
46    IF branch == zero GOTO 18;
47    branch = branch − one;
48    IF branch == zero GOTO 31;
49 height = height − one ;                  /* subtract one for output height */
50 M[stack_ptr] = height;
51 output_ptr = stack_ptr;
52 output_len = 1;
53 HALT
```

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

4 The Word-RAM Model

One 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: Use "bignum" arithmetic, where we write our numbers as an array of digits in base 2^w (similarly to what you did in PS1 for Radix Sort with a large base b). So an n-bit number is now expressed as an array of $m = \lceil n/w \rceil$ words Using the grade-school algorithm for multiplication, we can multiply two such numbers using $O(m^2)$ word-operations. There are asymptotically faster methods for multiplying large integers, such as Karatsuba's algorithm (taught in CS124), which has runtime $O(m^{\log_2 3})$ and ones based on the Fast Fourier Transform that have runtime $O(m \log m)$.[1]

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: If we do this, everything is a constant - we can't deal with arbitrarily sized inputs. Then our entire theory of algorithms doesn't make sense!

[1] Achieving this runtime is a development from just two years ago, resolving a 40-year old conjecture! https://www.jstor.org/stable/10.4007/annals.2021.193.2.4

Definition 4.1. 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: array of length S, with entries in $\{0, 1, \ldots, 2^w - 1\}$,
- Operations: are redefined from RAM Model to always produce outputs in that range (e.g. taking the result modulo 2^w)
- Initial settings: When a computation is started on an input x, which is an array consisting of n natural numbers, the memory size is taken to be $S = n$, and word length is taken to be $w = \lceil \log_2 \max\{S, x[0], \ldots, x[n-1]\} \rceil$.
- Increasing S and w: If the algorithm needs to increase its memory size beyond S, it can issue a MALLOC command, which increments S by 1 and if $S > 2^w$, it also increments 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.

In general, in this class, 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 4.1. 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.

Other Kinds of Numbers. Our definition of the RAM and Word-RAM Model only includes natural numbers, and in particular does not allow for real numbers, in contrast to the way we have been presenting our computational problems so far (e.g. allowing sorting keys to be arbitrary real numbers). Here's how we can represent numbers beyond $\mathbb{N}$:

- (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.) Python and other programming languages often us *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.

Operations on Words. 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 linearly 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).