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

Section 3: Models of Computation

Harvard SEAS - Fall 2023 *Sept. 28, 2023*

1 Introduction

Up to this point, we've reasoned about runtimes from a relatively informal perspective, either in terms of the physical amount of time a program takes to run or the number of operations it executes. Neither of these is satisfactory: multiple runs of the same program may take different amounts of time, which many of you observed when working on problem set 1, and we didn't specify what we meant by an "operation"—even the asymptotic number of operations a program needs may depend on what qualifies as a basic operation.

The RAM Model formalizes our definition of programs and allows us to rigorously define runtime. While the RAM Model may seem slightly different from Python or other programming languages, we can prove they can compute the same problems. The upshot is that we can reinterpret the questions of runtime and computability in higher level algorithms in terms of this simpler, lower-level model.

2 The RAM Model

Recall the definition of a RAM program from lecture:

Definition 2.1. 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) $\texttt{var} = c$, for a variable $\texttt{var} \in V$ and a constant $c \in \mathbb{N}$.
- (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 $+$, $-$, $\times$, $/$.
- (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 $k \in \{0, 1, \ldots, \ell\}$.

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: semantics)**.** 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 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 ℓ is reached (possibly due to a GOTO ℓ), the output $P(x)$ is defined to be the subarray of M of length `output_len` starting at location `output_ptr`. That is,

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

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 ∞).

Question 2.3. What would be the value of `output_ptr` and `output_len` for some RAM program that takes in an array of size `input_len` and sorts it in place, assuming `input_ptr` $= 0$?

2.1 Synthesizing notes on RAM

- Each of the command types is defined as *one operation*, and one operation corresponds to one line in the RAM program. Now that we have crystallized our definition of operations, we can now count *exactly* how many operations a program takes by the number of commands it executes. You'll practice this computation in section and in the problem set!

- The set of temporary variables V is fixed no matter what the input, whereas the memory M can grow. In lecture, we showed that we can simulate a program with any fixed set of variables using only 9 variables by offloading their values onto the memory.

- In Problem Set 3, you will write a Python program to simulate RAM programs. We gave you the scaffolding of operations, and your job is to fill them in. You shouldn't have to write more than 20 lines of code to complete the simulator.

Question 2.4. Consider the RAM program below.

```
   Input     : A single natural number N (as an array of length 1)
   Output    : A mystery...
   Variables: input_len, output_ptr, output_len, counter, result, zero, one
 1 zero = 0;
 2 one = 1;
 3 output_len = 1;
 4 output_ptr = 0;
 5 result = 1;
 6 counter = M[zero];
 7     IF counter == 0 GOTO 11;
 8     result = result * counter;
 9     counter = counter - one;
10     IF zero == 0 GOTO 7 ;
11 M[output_ptr] = result ;
```

- What does the algorithm do?

- Compute the runtime (that is, number of operations) as an exact function of N.

Question 2.5. (***Pseudocode to RAM**) [1] Write a RAM Program to output the natural numbers 1 through 100, leaving out multiples of 3. Use the following pseudocode as a guide:

```
1 NaturalNumbersNoMultThree()
  Input  : None
  Output : All x ∈ [1, 100] : x  mod 3 ≠ 0
2 output_arr = [0] × output_len;
3 memory_slot = 0;
4 foreach i ∈ [1, 100] do
5 |   if ⌊i/3⌋ · 3 ≠ i then
6 |   |   output_arr[memory_slot] = i;
7 |   |   memory_slot = memory_slot + 1;
8 return output_arr
```

[1] For TFs: Starred problems (*) are great additional practice problems to point students to if you cannot get to these during section time.

3 Motivating the Word RAM Model

The Word RAM model allows us to complete additional bookkeeping to make sure that our single operations are bounded. That way, we can account for the additional work of performing operations on larger inputs instead of claiming that they are equivalent.

Definition 3.1. 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 $\{0, 1, \ldots, 2^w - 1\}$. Reads and writes to memory locations larger than S have no effect.
- **Operations:** Addition and multiplication are redefined from RAM Model to return $2^w - 1$ if the result would be $\geq 2^w$.[2]
- **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 $w = \lfloor \log \max\{S, x[0], \ldots, x[n-1]\} \rfloor + 1$. (This setting is to ensure that $S, x[0], \ldots, x[n-1]$ are all strictly smaller than 2^w and hence fit in one word.)
- **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, sets $M[S-1] = 0$, 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`.

Question 3.2. Why is the word length initially set to $w = \lfloor \log \max\{S, x[0], \ldots, x[n-1]\} \rfloor + 1$?

Question 3.3. Write down the size of `mem_size` and the size of `word_len` after completing the following steps:

- Input of length $N = 7$: `[1, 5, 0, 6, 2, 7, 4]`
- MALLOC

3.1 Synthesizing notes on Word-RAM

- The reason why we keep track of word size is to bound exactly how many bits are allowed as an input to an operation. Otherwise, we could perform additions, multiplications, memory assignments, etc. on inputs of arbitrary size and call all of that a single operation! In our model, word size adjustments are handled automatically when we increment the memory. (However, if we were to implement this in real life, they would have a runtime.)

[2] A more standard choice is for the result to be returned mod 2^w, but we instead clamp results to the interval $[0, 2^w - 1]$ (known as *saturation arithmetic*) for consistency with how we defined subtraction in the RAM Model. If all arithmetic is done modulo 2^w, then the Conditional GOTO should also be modified to allow the condition to be an inequality, since we can no longer use the subtraction to simulate inequality tests.

- We might define the runtime of operations in terms of the word length. However, as we've seen, we usually define the word length in terms of something else, like the length of the input.

3.2 Simulating Word-RAM vs. RAM

In lecture, we saw that we can simulate any RAM program with a Word-RAM Program, and vice versa.

Theorem 3.4. *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*

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

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

$$T_{P'}(x) = O\left(T_P(x) + n + w_0\right),$$

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

4 Expressivity

In this unit, we built up the foundation that Python, RAM, assembly, etc. all run within a constant factor of one another. If we show that these models of computation are equivalent, we can use this general model to make claims about whether problems can be computed at all.

Example problem structure: Given a program P with a fancy new operation, you will show that an ordinary (Word-)RAM program P' can simulate it under a certain runtime. We often will aim to show that the runtime of P will be within a constant factor of the runtime of P', but depending on the operation that bound may differ.

Proof strategy

1. **Computability (operation):** Derive a way for the ordinary RAM program to simulate the exact operation. In other words, create the new operation out of old operations.

[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 $S = O(n + T_P(x))$ 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. **Runtime (operation):** Given a fancy new operation, calculate the runtime of the equivalent operations if we had to recreate the new operation under the old model.

3. **Computability (program):** Deduce that the old model can compute the same problems as the new model (for every instance of the new command, substitute in your translation!)

4. **Runtime (program):** Express the runtime of P' as a function of the runtime of P.

Question 4.1. (MOVE operation) One common assembly language command that we didn't include in our (Word-)RAM model is a MOVE operation, which directly copies a value from one memory location to another. In our RAM notation, this would have the syntax $M[\texttt{var}_i] = M[\texttt{var}_j]$ for variables $\texttt{var}_i$, and $\texttt{var}_j$. Prove that for every MOVE-augmented (Word-)RAM program P, there is an ordinary (Word-)RAM program P' such that for every input x, $P'(x) = P(x)$ and $T_{P'}(x) = O(T_P(x))$.

Question 4.2. (***Other Word Operations**) Suppose we have an Log-Augmented-RAM model that has a `log_2` operation, which computes $\lfloor \log_2(x) \rfloor$ for a given integer x. Given an integer x that can be represented in w bits as $x = x_{w-1}x_{w-2}\cdots x_0$, prove that the `log_2` operation can be computed in our Word-RAM model (without an `log` operation) in time $O(w)$. Feel free to introduce additional temporary variables.

Question 4.3. (***XOR Word Operations**) One common assembly language operation that we did not include in our Word-RAM model is *bitwise-XOR*. Given w-bit numbers x and y with binary representations $x = x_{w-1}x_{w-2}\cdots x_0$ and $y = y_{w-1}y_{w-2}\cdots w_0$, their bitwise XOR $z = x \oplus y$ is the number whose binary representation $z = z_{w-1}z_{w-2}\cdots z_0$ satisfies $z_i = x_i \oplus y_i$ for $i = 0, \ldots, w-1$.

1. Prove that when the current word size is w, a bitwise XOR operation $z = x \oplus y$ can be computed in our Word-RAM model (without an $\oplus$ operation) in time $O(w)$. Here you are given x, y, z as individual variables in the Word-RAM program (no need to read from or write to memory), and the current word size is also given as a variable `word_len`. You may introduce additional temporary variables if useful.

2. Using Part 1, show that for every XOR-extended Word-RAM program P, there is an ordinary Word-RAM Program P' such that for every input x, $P'(x) = P(x)$. As above, you may assume that the current word size is given as a variable `word_len`, which is automatically updated as the word size increases.

   In addition, argue that when the input $x = (x[0], \ldots, x[n-1])$ satisfies $x[0], x[1], \ldots, x[n-1] \leq n$ (i.e. the input numbers are not too big relative to the length) and $T_P(x) \geq n$ (i.e. P runs in enough time to read the entire input), then

   $$T_{P'}(x) = O\left(T_P(x) \cdot \log T_P(x)\right).$$

   (Hint: first show that the maximum memory size used by P on x is at most $n + T_P(x)$.)

The take-away point is that while the exact choice of which operations to include in a RAM model may affect the asymptotic running time, it typically affects it by logarithmic factors (so it might make the difference between $O(n)$ and $O(n \log n)$, but not between $O(n)$ and $O(n^2)$).

5 General Programs

Theorem 5.1 (informal)**.** [5]

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. Compilers! Our computers are not built to directly run programs in high-level programming languages like Python. Rather, high-level programs are *compiled* into assembly language, which is then (fairly directly) translated into machine code that is run by the CPU.[6] In assembly language, what we are calling *variables* are referred to as *registers*, and these represent actual physical storage locations in the CPU.)

2. You will see this in Problem Set 3. 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.

□

[5] 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.

[6] Python is an interpreted rather than compiled language. Python programs (or rather their bytecode) are actually executed by an interpreter that was originally written in C (or Java) but is compiled to assembly language in order to run.