CS1200: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Section 3: Models of Computation

Harvard SEAS - Fall 2025 *Sept. 25, 2025*

1 Problems

Question 1.1. (**RAM Model**) 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` $= 0$?

Question 1.2. (**RAM Model**) 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 1.3. (**RAM Model**)

(*****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:

```
NaturalNumbersNoMultThree()
Input    : None
Output   : All $x \in [1, 100] : x \mod 3 \neq 0$
$output\_arr = [0] \times$ output_len;
$memory\_slot = 0$;
foreach $i \in [1, 100]$ do
    if $\lfloor i/3 \rfloor \cdot 3 \neq i$ then
        $output\_arr[memory\_slot] = i$;
        $memory\_slot = memory\_slot + 1$;
return $output\_arr$
```

Question 1.4. (Motivating the RAM Model) Why might $T_{P[w]}(x)$ be different across different values of w, and why does the maximum runtime represent the "true" runtime?

Question 1.5. () (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 $T_{P'}(x) = O(T_P(x))$.

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

2 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.

3 The RAM Model

Recall the definition of a RAM program from lecture:

Definition 3.1. 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)
- (arithmetic)
- (read from memory)
- (write to memory)
- (conditional goto)

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

Definition 3.2 (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 ∞).

```
**Input**      : None
**Output**     : All x $\in [1, 100], x \mod 3 \neq 0$
**Variables**  : input_len, output_ptr, output_len, counter, zero, one, three,
                 one_hundred_one, checker
1  zero = 0 ;                                            /* useful constants */
2  one = 1;
3  three = 3;
4  one_hundred_one = 101;
5  output_ptr = 0 ;                             /* output begins at start of memory */
6  output_len = 0;
7  counter = 1;
8      checker = one_hundred_one − counter ;                   /* start of loop */
9      IF checker == 0 GOTO 19;
10     checker = counter ÷ three;
11     checker = checker × three;
12     checker = counter − checker;
13     IF checker == 0 GOTO 16;
14     $M$[output_len] = counter ;                          /* non multiple of 3 */
15     output_len = output_len + one;
16     counter = counter + one;
17     IF zero == 0 GOTO 8 ;                                     /* end of loop */
18 /* We only write "HALT" after the last line of a program as a reminder that
      a program halts if it reaches line $\ell$.  This line $\ell = 19$ could also be
      blank in this solution.  */
19 HALT ;
```

4 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 4.1. The *Word RAM Model* is defined like the RAM Model except that it has a *word length* w and *memory size* S. w is static, meaning that it is set to an arbitrary value in $\mathbb{N}$ prior to running the program with input x and unchanging during the computation. On the other hand, S is dynamic and able to change during the computation. We describe how these parameters are used:

- 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. Initially $S = n$, the length of the input array x. Additional memory can be allocated via a MALLOC operation, which increments S by 1.
- Output: if the program halts, the output is defined to be $(M[\texttt{output\_ptr}], M[\texttt{output\_ptr} + 1], \ldots, M[\min\{\texttt{output\_ptr} + \texttt{output\_len} - 1, S - 1\}])$. That is, portions of the output outside allocated memory are ignored.
- Operations: Addition and multiplication are redefined from RAM Model to return $2^w - 1$ if the result would be $\geq 2^w$.[2]
- Crashing: A Word-RAM program *crashes* on input x and word length w if any of the following happen during its computation:
    1. If $c_j \geq 2^w$ for some j, that is, one of the constants appearing the variable assignments in program P is at least 2^w.
    2. If $x[i] \geq 2^w$ for some $i \in [n]$
    3. $S > 2^w$. (This can happen because $n > 2^w$, or if when $2^w + 1 - n$ MALLOC commands are executed.)

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.

Definition 4.2. 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. For every word length w, $P[w](x)$ halts (either producing an output or crashing).

[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.

2. For all sufficiently large word lengths $w \in \mathbb{N}$, $P[w](x)$ does not crash.

3. 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 $P[w](x) = \bot$ if $f(x) = \emptyset$.

Below is an example Word-RAM program. This algorithm is meant to duplicate an array x twice, with $x \circ x$ being two copies of x concatenated together.

```
Duplicate($x$):
Input      : An array $x$ of length $n$
Output     : The array $x \circ x$, 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 4.1: A Word-RAM program to illustrate the use of MALLOC and the output truncation. At the end of the program, $S = 2n$ but output_len $= 3n$, so the output is truncated to end at $M[S-1]$.

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). However, we also want to define the runtime of a program on an input independent of the word size being used, so we also introduce the following:

Definition 4.3. The *running time* of a word-RAM program P on an input x is defined to be

$$T_P(x) = \max_{w \in \mathbb{N}} T_{P[w]}(x).$$

4.1 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 4.4. *Let $\Pi = (\mathcal{I}, \mathcal{O}, f)$ be a computational problem, with some fixed encoding of elements of $\mathcal{I}$ as arrays of natural numbers. Let $T : \mathcal{I} \to \mathbb{R}^+ \cup \{\infty\}$ such that for all $x \in \mathcal{I}$, $T(x) \geq$*

$n+2$,[3] *where n as the length of the array encoding x, and every entry of that array has bitlength $O(\log T(x))$. Then the following are equivalent:*

1. *Π can be solved by a Word-RAM program P with $T_P(x) = O(T(x))$.*

2. *Π can be solved by a RAM program Q with $T_Q(x) = O(T(x))$ that also satisfies the following conditions for all input arrays x:*

   (a) *All of the values that Q's variables and memory locations hold during its computation on x have bitlength $O(\log T(x))$.*

   (b) *The largest memory location accessed by Q or included in Q's output on input x is $O(T(x))$.*[4]

5 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.

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.

10

6 General Programs

Theorem 6.1 (informal). [5]

[3]The $+2$ is just to ensure that $\log T(x) \geq 1$ even when $n = 0$.

[4]This condition can be removed at a small cost in runtime; see Problem **??**.

[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.

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]20

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.

□

7 Church-Turing Thesis

Definition 7.1. We call a model *Turing–equivalent* if and only if it has the same computational power as a Turing machine. More specifically, if a computational problem Π is solvable by one of the following models of computation, then it is solvable in all of them:

- RAM programs
- %-extended RAM programs
- Python programs
- OCaml programs
- C programs (modified to allow a variable/growing pointer size)
- Word-RAM programs
- XOR-extended RAM or Word-RAM programs
- Lambda calculus (an early model of computation invented by the mathematician Alonzo **Church**)
- Turing machines (an early model of computation invented by the mathematician Alan **Turing**)
- $\vdots$

[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. In assembly language, what we are calling *variables* are referred to as *registers*, and these represent actual physical storage locations in the CPU.)

This definition leads us to the Church–Turing Thesis, which has two variants.

1. Turing-equivalent models of computation capture the intuitive notion of an algorithm.
2. A computation can be done within our universe's laws of physics if and only if it can be done using Turing-equivalent models.

7.1 Extended Church-Turing Thesis

Extended Church–Turing Thesis v2: Every physically realizable, deterministic, and sequential model of computation can be simulated by a Word-RAM program with only a polynomial slowdown in runtime. Conversely, Word-RAM programs can be physically simulated in a deterministic and sequential manner in real time only polynomially slower than their defined runtime.

Like the basic Church–Turing Thesis, the Extended Church–Turing Thesis was originally formulated for Turing Machines, as discussed in Chapter 8. Turing Machines and Word-RAM Programs are polynomially equivalent, meaning that they can simulate each other with a polynomial slowdown. Any computational model that is polynomially equivalent to these is referred to as a strongly Turing-equivalent model of computation. For this reason, the Extended Church–Turing Thesis is also often called the Strong Church–Turing Thesis