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

Lecture 6: The RAM Model

Harvard SEAS - Fall 2025 *2025-09-18*

1 Announcements

- Salil Zoom OH Mon, tentatively 2-2:45pm (check course Google calendar).
- Update on posting of Computational Models part of textbook

2 Recommended Reading

- Hesterberg–Vadhan, Chapter 5
- CLRS Sec 2.2

3 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 the number of "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. In approximate order from "far from physical" to "close to physical," some examples of computational models are:

0. Natural language (e.g. 'calculate the prime factorization of the following number'):
1. Declarative and functional programming languages:
2. Imperative high-level programming languages:
3. "Close to the metal" imperative programming languages:
4. Architecture-level models:
5. Hardware models:

Translation from higher-level to lower-level models:
Also need a complexity measure to model "time" (or other resources).
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 .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 .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.

Our focus for the rest of this chapter 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.

5 Iterative Algorithms

Theorem .3. *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 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 */
```

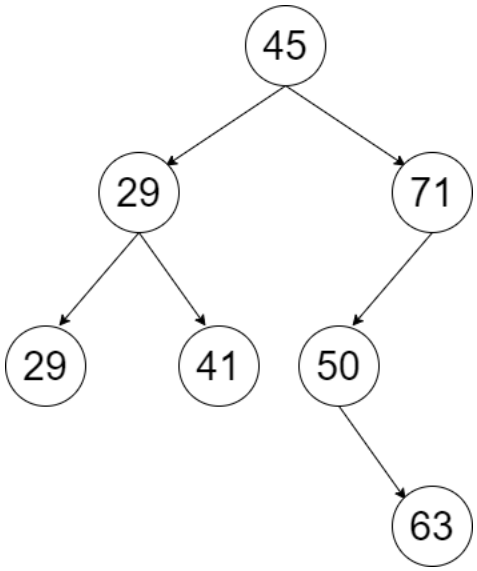
Algorithm .1: RAM implementation of `InsertionSort()`

6 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:
- Rational numbers:
- Real numbers:
- Strings:

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



Assuming all of the associated values are 0, this would be represented as the following array of length 28:

$$[45, 0, 4, 8, 29, 0, 12, 16, 71, 0, 20, 0, 29, 0, 0, 0, 41, 0, 0, 0, 50, 0, 0, 24, 63, 0, 0, 0]$$

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 of 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.

7 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 to implement a *call stack*. We simulate a function call $f(x)$ through the following steps:

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

Because many things can be pushed onto the stack, this call stack can implement nested recursive calls.

Below we present an example of a recursive computation of the height of a binary tree. Since our RAM model doesn't allow negative numbers, but an empty tree has height -1, our recursive functions will compute the height plus one, and we will subtract one from the "height" output at the end to get the true height. 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.

```
   CalculateHeight(T):
   Input      : A Binary Tree T of Key-Value Pairs, given as an array of 4-tuples
                (K, V, P_L, P_R)
   Output     : The height of T
 0 zero = 0 ;                                              /* useful constants */
 1 one = 1;
 2 two = 2;
 3 stack_ptr = input_len + zero;
 4 M[stack_ptr] = zero ;        /* push pointer to root of tree to top of stack */
 5 stack_ptr = stack_ptr + one;
 6 M[stack_ptr] = zero ;                      /* branch-indicator for root call */
 7    branch = M[stack_ptr] ;                            /* pop branch indicator */
 8    stack_ptr = stack_ptr − one;
 9    node_ptr = M[stack_ptr] ;                  /* pop pointer to current node */
10    stack_ptr = stack_ptr + two ;          /* and repush both back onto stack */
11    temp_ptr = node_ptr + two;
12    child_ptr = M[temp_ptr] ;                        /* 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 7 ;                                          /* recurse */
```

Algorithm .2: RAM implementation of `CalculateHeight()`

```
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 `CalculateHeight()`

8 Reductions in the RAM model

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

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

which is an instruction to call the oracle on the array $(M[\texttt{var}_0], M[\texttt{var}_0+1], \ldots, M[\texttt{var}_0+\texttt{var}_1-1])$ and write the oracle's answer in the locations $(M[\texttt{var}_2], M[\texttt{var}_2+1], \ldots)$.

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

```
IntervalSchedulingViaSorting(x):
Input  : An array x = (a_0, b_0, a_1, b_1, ..., a_{n-1}, b_{n-1}), occupying memory locations
         M[0], ..., M[2n-1], with a_i ≤ b_i for all i
Output : 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