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

Lecture 6: The RAM Model

Harvard SEAS - Fall 2022 *Sept. 20, 2022*

1 Announcements

Recommended Reading:

- CLRS Sec 2.2
- **Thursday class 9-10:15am in Cambridge, Emerson 105**
- Adam OH today after class.
- PS2 due tomorrow, Participation Highlights 1 due Sat, PS0 revisions due Sun.
- Fill out reflection survey after SRE.
- Remember to bring/use name placards

PS0+PS1 Survey feedback:

- Time spent on psets is too long!
  1. PS0 median time 10hrs (cf. 14hrs in 2021), 25% spent 15+hrs (cf. 18+hrs in 2021).
  2. PS1 median time 12hrs (cf. 14hrs in 2021), 25% spent 20+hrs (cf. 19+hrs in 2021).
  3. Stop if you are spending so long! Remember revision policy, and up to 4 R- grades on psets is still in A-range.
  4. It will get easier! Last year Q eval: median time 10hrs/week, 25% spent 14+hrs/week.
  5. Look through entire pset before beginning
  6. Google sheet for finding pset buddies
  7. Pset due dates: can't change at this point, Fri-Fri or overlapping cycles bring other complications (weekend sections, same due date as other courses, scheduling of section and OH).
- Office hours: not enough and too crowded
  1. We were understaffed after the class tripled in size.
  2. We have hired a bunch of new part-time CAs for grading and OH!
- Lectures
  1. Mixed calls for more proof details, more examples, more big picture. We will try to strike a balance! One of the most important roles for lecture in cs120 is to introduce you to the mathematical abstractions. Section is a good place for more examples and going over proof details.

2. Pacing uneven and a bit fast. Will try to smooth it out, need to balance against requests for lecture to do more.
3. Writing. Legibility better in person. Still bad on Zoom/Panopto; can't do much about that.
4. Lecture Notes. We can post *drafts* of the detailed lecture notes on the course schedule in advance, but (a) strongly encourage you to work from the un-detailed ones during class in order to be actively engaged, and (b) note that the detailed lecture notes will usually be updated with corrections and more details after class.
5. In-class activities (e.g. edstem quizzes) appreciated.

- Sections: very positive feedback. Attendance encouraged!

2 Goals

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 a basic operation? To address this, we need to specify a *computational model* for describing algorithms.

What do we want from a computational model?

- Unambiguity.
- Expressivity.
- Mathematical simplicity.
- Robustness.
- Technological relevance.

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)**.** A *RAM Program* $P = (V, C_0, \ldots, C_{\ell-1})$ consists of a finite set V of *variables* (a.k.a. *registers*), and a sequence $C_0, C_1, \ldots, C_{\ell-1}$ of commands, 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 3.2 (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:

2. Execution:

3. Output:

The *running time* of P on input x is defined to be:

The definition of the RAM Model above is mathematically precise, so achieves our unambiguity desideratum (unless we've forgotten to specify something!). 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 the other desiderata next time.

4 Iterative Algorithms

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

```
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
1  zero = 0 ;                                                   /* useful constants */
2  one = 1;
3  two = 2;
4  output_ptr = 0 ;                                     /* output will overwrite input */
5  output_len = input_len + zero;
6  outer_key_ptr = 0 ;                             /* 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 = 0 ;                      /* 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_value = M[temp_ptr] ;                       /* value to be inserted */
20             temp_key = M[inner_key_ptr] ;               /* start of 2nd inner loop */
21             temp_ptr = inner_key_ptr + one;
22             temp_value = M[temp_ptr];
23             M[inner_key_ptr] = insert_key;
24             M[temp_ptr] = insert_value;
25             insert_key = temp_key + zero;
26             insert_value = temp_value + zero;
27             inner_key_ptr = inner_key_ptr + two;
28             IF inner_rem == 0 GOTO 8;
29             inner_rem = inner_rem − one;
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 ;                                                    /* not an actual command */
```

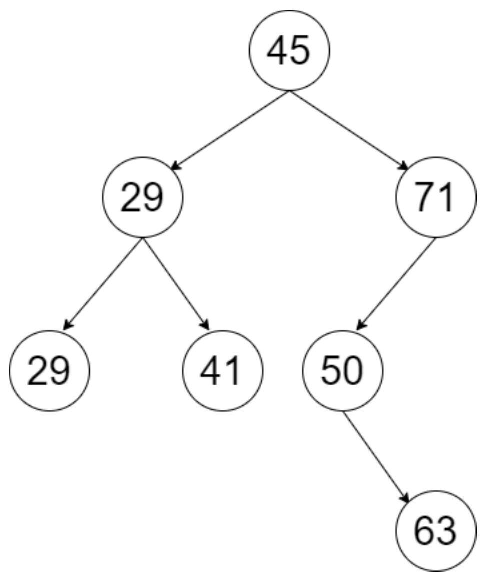
Algorithm 1: RAM implementation of Insertion Sort

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:
- 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_0, V_0, 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:

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.

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

```
zero = 0 ;                                   /* useful constants */
one = 1;
two = 2;
stack_ptr = input_len + zero;
M[stack_ptr] = zero ;    /* push pointer to root of tree to top of stack */
stack_ptr = stack_ptr + one;
M[stack_ptr] = zero ;                  /* branch-indicator for root call */
    branch = M[stack_ptr] ;                     /* pop branch indicator */
    stack_ptr = stack_ptr − one;
    node_ptr = M[stack_ptr] ;             /* pop pointer to current node */
    stack_ptr = stack_ptr + two ;      /* and repush both back onto stack */
    temp_ptr = node_ptr + two;
    child_ptr = M[temp_ptr] ;                   /* pointer to left child */
    IF child_ptr == 0 GOTO 23;
    M[stack_ptr] = child_ptr ;            /* push pointer to left child */
    stack_ptr = stack_ptr + one;
    M[stack_ptr] = one ;                        /* left branch indicator */
    IF zero == 0 GOTO 8 ;                                     /* recurse */
```

Algorithm 2: RAM implementation of Calculate Height

```
19    left_height = M[stack_ptr] ;                    /* pop height of left child */
20    stack_ptr = stack_ptr − one;
21    node_ptr = M[stack_ptr] ;                    /* pop pointer to current node */
22    IF zero == 0 GOTO 24
23    left_height = 0 ;                                  /* left child is empty */
24    temp_ptr = node_ptr + three;
25    child_ptr = M[temp_ptr] ;                       /* pointer to right child */
26    IF child_ptr == 0 GOTO 38;
27    M[stack_ptr] = left_height ;                   /* push height of left child */
28    stack_ptr = stack_ptr + one;
29    M[stack_ptr] = child_ptr ;                   /* push pointer to right child */
30    stack_ptr = stack_ptr + one;
31    M[stack_ptr] = two ;                            /* right branch indicator */
32    IF zero == 0 GOTO 8 ;                                           /* recurse */
33    right_height = M[stack_ptr] ;                 /* pop height of right child */
34    stack_ptr = stack_ptr − one;
35    left_height = M[stack_ptr] ;                    /* pop height of left child */
36    stack_ptr = stack_ptr − one;
37    IF zero == 0 GOTO 39;
38    right_height = 0
39    branch = M[stack_ptr] ;                             /* pop branch indicator */
40    diff_heights = left_height − right_height;
41    IF diff_heights == 0 GOTO 44 ;                         /* right-child taller */
42    height = left_height + one ;                            /* left-child taller */
43    IF zero == 0 GOTO 45;
44    height = right_height + one;
45    IF branch == zero GOTO 51;
46    M[stack_ptr] = height ;                                /* push return value */
47    branch = branch − one;
48    IF branch == zero GOTO 19;
49    branch = branch − one;
50    IF branch == zero GOTO 33;
51 height = height − one ;                  /* subtract one for output height */
52 M[stack_ptr] = height;
53 output_ptr = stack_ptr;
54 output_len = 1;
55 HALT
```

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

7 General Programs

Theorem 7.1. *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.

2.

□

8 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 8.1. An *oracle-aided RAM Program* is like an ordinary RAM program, except it can also have commands of the form

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

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

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

Input : An array $x = (a_0, b_0, a_1, b_1, \ldots, a_{n-1}, b_{n-1})$, occupying memory locations $M[0], \ldots, M[2n-1]$, with $a_i \leq b_i$ for all i

Output : 1 (YES) if all of the intervals $[a_i, b_i]$ are disjoint, 0 (NO) otherwise

```
1  zero = 0;
2  one = 1;
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 − one ;       /* how many adjacent pairs left to check */
9      IF remaining == 0 GOTO 19;
10     = 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−;
14     IF temp == 0 GOTO 18 ;                                      /* conflict found */
15     temp_ptr = temp_ptr + one;
16     remaining = remaining − one;
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