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

Problem Set 3

Harvard SEAS - Fall 2022 *Due: Wed Sept. 28, 2022 (11:59pm)*

Your name: Aika Aldayarova
Collaborators: Christine Lee
No. of late days used on previous psets:
No. of late days used after including this pset:

The purpose of this problem set is to solidify your understanding of the RAM model (and variants), and the relations between the RAM model, the Word-RAM model, Python programs, and variants. In particular, you will build skills in simulating one computational model by another and in evaluating the runtime of the simulations (both in theory and in practice).

1. (Simulation in practice: RAMs on Python) In the Github repository, we have given you a partially written Python implementation of a RAM Model simulator. Your task is to fill in the missing parts of the code to obtain a complete RAM simulator. Your simulator should take as input a RAM Program P and an input x, and simulate the execution of P on x, and return whatever output P produces (if it halts). The RAM Program P is given as a Python list $[v, C_0, C_1, \ldots, C_{\ell-1}]$, where v is the number of variables used by P. For simplicity, we assume that the variables are named $0, \ldots, v-1$ (rather than having names like "tmpptr" and "insertvalue"), but you can introduce constants to give names to the variables. The 0^{th} variable will always be `input_len`, the 1^{st} variable `output_ptr`, and the 2^{nd} variable `output_len`. A command C is given in the form of a list of the form [`cmd`], [`cmd`, i], [`cmd`, i, j], or [`cmd`, i, j, k], where `cmd` is the name of the command and i, j, k are the indices of the variables or line numbers used in the command. For example, the command $\texttt{var}_i = M[\texttt{var}_j]$ would be represented as ("`read`", i, j). See the Github repository for the precise syntax as well as some RAM programs you can use to test your simulator.

2. (Empirically evaluating simulation runtimes and explaining them theoretically)

   Consider the following two RAM programs:

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

```
   Input     : A single natural number N (as an array of length 1)
   Output    : 11^(2^N+1) mod 2^32 (as an array of length 1)
   Variables: input_len, output_ptr, output_len, counter, result, temp, W
 0 zero = 0;
 1 one = 1;
 2 eleven = 11;
 3 output_len = 1;
 4 output_ptr = 0;
 5 result = 11;
 6 W = 2^32;
 7 counter = M[zero];
 8     IF counter == 0 GOTO 15;
 9     result = result * result;
10     temp = result/W;
11     temp = temp × W;
12     result = result - temp;
13     counter = counter - one;
14     IF zero == 0 GOTO 8;
15 result = result * eleven;
16 temp = result/W;
17 temp = temp × W;
18 result = result - temp;
19 M[output_ptr] = result;
```

(a) Exactly calculate (without asymptotic notation) the RAM-model running times of the above algorithms as a function of N. Which one is faster?

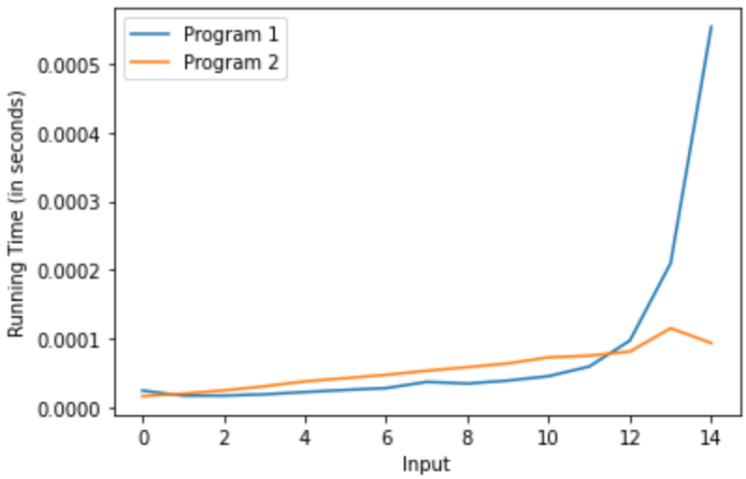
The first algorithm's running time is $4n + 10$. Specifically, the first algorithm's first 7

lines and last 2 lines, together, make up the trailing 9 in the runtime. Lines 7 through 10 (4 lines total) form the loop of the algorithm that is run "input_len" or "n" number of times. At the end of the final loop, line 10 redirects the algorithm back to line 7 to check once more that the condition is not met, after which the algorithm jumps to line 11. This additional check of the condition results in 1 more line being accounted for, thus adding 1 onto 9. Summing both the $4n$ and 10 together, we get our first algorithm's running time.

The second algorithm's running time is $7n + 14$. The explanation of the runtime of this algorithm follows that of the first algorithm by symmetry. The term $7n$ comes from repeating the 7 lines within the code's loop "n" times, and the term 14 comes from running the first 8 lines, the last 5 lines, and the extra line within the loop together.

Thus, the first algorithm has a faster running time of the two.

(b) Using your RAM Simulator, run both RAM programs on inputs $N = 0, 1, 2, \ldots, 15$ and graph the actual running times (in clock time, not RAM steps). (We have provided you with some timing and graphing code in the Github repository.) Which one is faster?



As seen in the graph attached above, the second program ends up being faster than the first, since it grows linearly while the first program grows exponentially.

(c) Explain the discrepancies you see between Parts 2a and 2b. (Hint: What do we know about the relationship between the RAM and Word-RAM models, and why is it relevant to how efficiently the Python simulation runs?)

Whereas in theory prog1 was faster, in practice prog2 was faster. The distinction between theory and practice is important because in theory, in RAM, every line/command takes $O(1)$ time. This is not true in practice since some lines of code may contain more complex commands than others that in reality would take longer to run. Additionally, in practice, variables can increase to significantly large values very quickly and what

may take one program just a few milliseconds to compute can take another program a few seconds to calculate.

Prog2 may be faster in reality because of the following points:

(1) This version of the algorithm deals with larger numbers more efficiently by utilizing modulus.

(2) Our computers are 64-bit machines and since the second program efficiently stores the large numbers, the number of operations per second our computers can compute using the second program is much faster than the first program.

These points are relevant to the language we're executing the code in because Python's runtime heavily depends on the size of the numbers it is calculating with. When our numbers get significantly high, we want our program to be able to have no trouble computing the answer in linear time, which is exactly the behavior of the second program.

(d) (optional[1]) Give a theoretical explanation (using asymptotic estimates) of the shapes of the runtime curves you see in Part 2b. You may need to do some research online and/or make guesses about how Python operations are implemented to come up with your estimates.

Was not able to attempt the problem due to lack of time!

3. (Simulating Word-RAM by RAM) Show that 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

$$\text{Time}_{P'}(x) = O\left(\text{Time}_P(x) + n + w_0\right),$$

where n is the length of x and w_0 is the initial word size of P on input x. (This was stated without proof in Lecture Notes 7.)

Your proof should use an *implementation-level* description, similar to our proof that RAM programs can be simulated by ones with at most c registers. Recall that Word-RAM programs have read-only variables `mem_size` and `word_len`; you may want to start your simulation by calculating these variables as well as another variable representing $2^{\texttt{word\_len}}$. Then think about how each operation of a Word-RAM program P can be simulated in a RAM program P', taking care of any differences between their semantics in the Word-RAM model vs. the RAM model. Don't forget MALLOC!

Proof. In order to simulate program P in terms of P' (which is a RAM program), we need to think back to our definition of a RAM program - definition 2.2 from section 3 notes. A RAM program consists of 3 steps: initialization, execution, and output, and we must be able to represent program P in terms of all of these steps. Thus, let's approach this simulation

[1]This problem won't make a difference between N, L, R-, and R grades.

one step at a time, and talk about each step's runtime one sub-step at a time:

(1) Initialization: First we must encode input x into the memory array; since the input is of length n, the runtime for this sub-step would be $O(n)$.

Then, specific to Word-RAM, we must calculate mem_size (referred to as S from now on); since we already know the length of the input, this sub-step will simply be a $O(1)$ time operation.

Next, specific to the Word-RAM as well, we need to calculate the word_len using the formula provided in the lecture 7 notes. The formula is the floor of $(log_2(max(S, x[0], x[1], ...))$. The sub-step of calculating the max of the set within the formula will take $O(n)$ time because we have to check every single term within the set to see if it might be the biggest value. Then, the step of taking the log of that max value will take $O(w_0)$ time because this operation focuses on extracting the number of bits we will need to express our numbers.

As a final sub-step within initialization, we need to calculate $2^{\texttt{word\_len}}$ because $2^{\texttt{word\_len}} - 1$ will be the maximum number we will be legally able to represent. This sub-step will be run in $O(1)$ time since it is a simple arithmetic operation.

Thus, the total runtime of the initialization step will be $O(n + w_0)$ time.

(2) Execution: In terms of simulating operations in Word-RAM, let us first dissect how the existing operations in RAM would be different when simulating Word-RAM:

Arithmetic operations would work similarly except their results would have to be capped to $2^{\texttt{word\_len}} - 1$. Since the operations itself would remain the same, they would take $O(1)$ time.

Reading and writing into memory would also function similarly except we would need to add conditionals to ensure we are indexing into valid spots in memory. If the read/write causes the system to access memory outside of the allowable bounds, it would be skipped. Because the operation itself would remain the same except additional if statements, it would take $O(1)$ time.

Assignment would also remain the same except checking to make sure the value a variable is being assigned to does not exceed the allowable bound. As such, its runtime would also be $O(1)$.

GOTO would not change except for potentially being shifted in the code, and as such, jumping to slightly shifted line numbers (it might be shifted in the code if changing the other operations causes us to add additional lines, which would shift the rest of the code). This operation would also take $O(1)$ time.

Now, let us think about Word-RAM-specific operations that we would need to append in our simulation using RAM: namely, the MALLOC operation.

In RAM, we simulate the operation of MALLOC by increasing either the mem_size or the word_len. We would increase the mem_size if we wanted to add extra, physical, allocated space within the memory array. We would increase the word_len if we wanted to increase the amount our existing memory indices could store. The time to do either of these operations would be $O(1)$ because all we are doing is increasing variable values by a certain amount, which is done in constant time.

Thus, the total runtime of the execution step would be $O(1) * Time_p(x)$. This is because none of the Word-RAM operations would be different in RAM, but still would take the RAM amount of time to complete.

(3) Output: Assuming that the output of P and P' is the same, the time of outputting the result would simply be $O(1)$.

All in all, our simulation changed significantly in the initialization step due to the need to simulate calculations of new, Word-RAM-specific variables. Thus, adding all of the runtimes of the substeps, we get a runtime of $O(Time_p(x) + n + w_0)$ of the simulation P'.

Q.E.D.