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: Nicole Chen
Collaborators: Jayden Personnat, Prin Pulkes, Dan Ennis, Sid Bharthulwar
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 $[\texttt{cmd}]$, $[\texttt{cmd}, i]$, $[\texttt{cmd}, i, j]$, or $[\texttt{cmd}, i, j, k]$, 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?

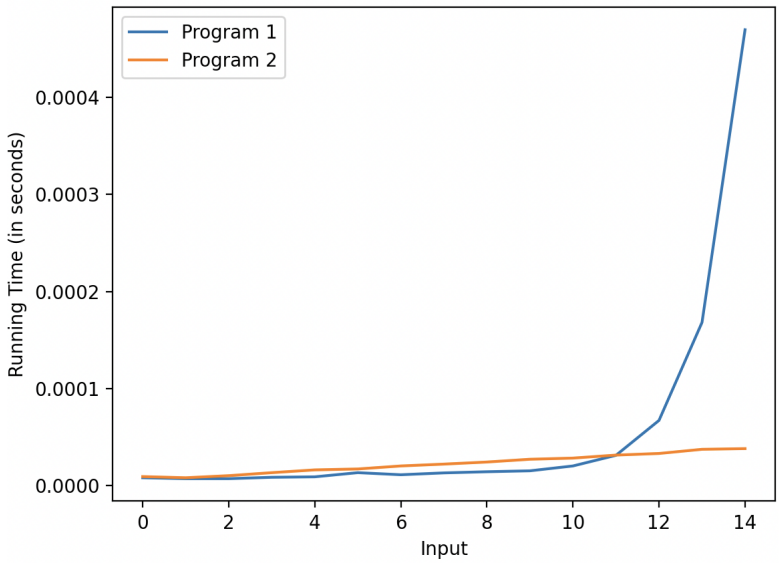
The first algorithm has $4N + 10$ run time. The second algorithm has $7N + 14$ run

time. We know that each line takes one time step. Therefore, we look at the run time analysis below:

For Algorithm 1, Lines 0 through 6 initializes variables and has a run time of 7. Lines 8-10 run N times because it iterates N times and Line 7 runs $N+1$ times because it is checked one more time. Therefore, Lines 7-10 run $4N+1$ time. Lines 11 and 12 have a run time of 2. This produces $4N+10$.

For Algorithm 2, Lines 0 through 7 initializes variables and has a run time of 8. Lines 9 through 14 run N time because it iterates N times and Line 8 runs $N+1$ times because it is checked one more time. Therefore, Lines 8 through 14 run $7N+1$ time. Lastly, Lines 15 through 19 has a run time of 5. Adding all of these, we get $7N+14$

(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?



Program 1 is initially faster. However, as input increases, Program 2 becomes significantly faster than Program 1.

(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?)

In the above graph in 2(b), we can see that the run times for both RAM programs begin to increase linearly. However, as the size grows, Program 1's run time grows considerably, unlike Program 2 which maintains a pretty linear run time as input increases. This is different from the run times we calculated in 2(a) and can be explained by analyzing the assumptions of RAM Models. First, each line of a RAM model is assumed to

run $O(1)$ time, which is not necessarily true in our actual Python RAM Model simulator. Namely, for operations such as `multiplication`, which the above two programs utilize, if we were to do those operations on extremely large numbers, these operations would not take $O(1)$ time as assumed. Therefore, the assumption that a line of RAM model runs $O(1)$ time does not hold true for our Python Simulator. For that reason, you can see why Program 1 does not follow the linear trajectory as expected from 2(a).

Now to reason why Program 1 starts off with similar run times as Program 2 but increases much more rapidly when input increases, we look at the algorithms themselves. For Program 1, *result* will reach 11^{2^N+1}. This is an extremely large number that takes considerably more run time to calculate with our defined operations than $O(1)$. This is slightly improved in Program 2. Although in Line 9 we are still multiplying *result* by *result*, we have variable *temp* that caps and "stifles" the value of *result* so that *result* in Program 2 will not grow as large as Program 1's *result*. In fact, the output of Program 2 is $11^{2^N+1} \mod 2^{32}$ which is less than the output for Program 1. As a result, because the variable values that operations such as `multiplication` are being utilized on in Program 2 are smaller than those in Program 1, it's run time will not increase as much for inputs 1 through 15 as compared to Program 1. One last thing is that because integers are capped at a certain value, the values of certain variables in our Python simulation may need to be represented as $BigNum$ rather than *integer* because those numbers grow too fast which may contribute to the longer run time.

Lastly, in the beginning, both Program 1 and Program 2 have run times that increase pretty linearly because for smaller inputs, the Python simulator should follow the RAM Model Algorithm run times, which are linear, because the assumption that each line of operations runs $O(1)$ holds for smaller inputs where multiplication, addition, subtraction, and division can run $O(1)$ time.

(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.

The distinction between the run time shapes of Problem 1 and Problem 2 is that one flows linearly from Inputs 0 to 15 while the other flows exponentially. We've analyzed in the 2 (c) that the difference comes from how large the variables and output are when operations such as multiplication are invoked. This is because of Python's decision to utilize the karatsuba algorithm for large numbers, where the karatsuba algorithm has $O(n^{1.59})$ for numbers of n bits.

We compare the programs mathematically by the karatsuba algorithm:

First, in program 1 the largest output derived from Line 8 is 11^{2n}. To find the number of bits in 11^{2n}, we do $\log(11^{2n})$. Therefore, to find the run time for the karatsuba

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

algorithm, we have:

$$\begin{aligned} & O(\log(11^{2n}) + 2)^{1.59} \\ = & O(2^n \cdot \log(11) + 2)^{1.59} \\ = & O((2^n)^{1.59} \cdot (\log(11) + \frac{1}{2^{n-1}})^{1.59}) \end{aligned} \tag{1}$$

As n approaches infinity, (1) reaches $O(2^n)^{1.59} \cdot c$ where c is a constant defined as $(\log 11)^{1.59}$. Therefore, our run time should look exponential, as is proven by the what the run time looks like in the graph for program 1.

On the other hand, in program 2, because our result is the mod of 2^{32}, it is always less than 2^{32}. Therefore, when multiplication is invoked on larger numbers in Lines 11 and 9, we know that the karatsuba method is capped at 2^{32}. Therefore, we find the run time for the karatsuba method to be:

$$\begin{aligned} & O(\log(2^{32})^{1.59} \\ = & O(32)^{1.59} \end{aligned} \tag{2}$$

Therefore, we can see that the run time during the karatsuba method for program 1 is considerably less than that of the exponential run time during the karatsuba method for program 2. Adding on additional lines of code, we therefore find that that program 2 grows linearly as the run time shows.

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!

Solution

- S' For starters, we modify P, the Word-RAM Model, so that the memory size S is stored in a variable S' that will be used for the RAM Model to access and update memory size. This takes $O(1)$ time because n is stored as a value in the variable `input_len` so we simply need to assign the value `input_len` into the variable S'.
- w_0' We also modify P, the Word-RAM Model, so that the initial word length w_0 is stored in a variable w_0' that will be used for the RAM Model to access and update word length. This takes $O(n + w_0)$ time. This is because w_0 is defined as

$\lfloor \log \max\{S, x[0], ..., x[n-1]\} \rfloor + 1$. Because our RAM Model does not have a command defined for `max`, in order to calculate that command, we must use existing operators. Therefore, supposed that P has memory M. We therefore define P' with M' where $M[i]$ will be represented by $M'[i]$ in P'. In order to calculate `max` for w'_0, we do as follows:

1. Calculate `max` by initializing a variable *temp* and read from $M'[input\_len - 1]$ so that $temp = M'[input\_len - 1]$.
2. Initialize a variable *counter* and assign it to $input\_len - 1$.
3. Initialize a variable *zero* and assign it to 0.
4. Implement a `goto` command for *counter* that jumps to step 10.
5. Implement a `subtract` command that takes the difference between *temp* and $M'[counter]$ and assigns that value to a new variable $difference$ (ie. $M'[counter] - temp$).
6. Implement a `goto` command for $difference$ that jumps to step 8.
7. Implement a `write` command that takes the value in $M'[counter]$ and writes it into the variable *temp*.
8. Implement a `subtract` command that takes the difference between *counter* and 1 and assigns that value to *counter*.
9. Implement a `goto` command for *zero* that jumps to step 4.
10. Additional Code

Therefore, the max command takes $O(n)$ time where n is the $input\_len$ of the input. This is because we must iterate through all n values of the input to determine the maximum value of the input values.

Let us say we find the maximum input value to be m. From there, we must determine $\log m + 1$. Because we have no command for log, we must utilize existing operators as follows:

1. Initialize a variable *zero* and assign it to 0.
2. Initialize a variable w'' and assign it to 0.
3. Implement a `goto` command for m that jumps to step 7.
4. Implement a `divide` command that divides 2 from m and assign that value to m.
5. Implement an `addition` command that adds 1 to w'' and assign that value to w''.
6. Implement a `goto` command for *zero* that jumps to step 3.
7. Additional Code

We therefore have the value w''. When we do $w'' + 1$, we get $\log m + 1$ which is w'_0. Adding 1 to w'' is just $O(1)$ because it is a basic addition operation. How many times do we iterate through the above lines of instructions to get w'_0? We iterate around $w_0 = w'_0$ times because $w'_0 - 1 = w_0 - 1 = \log m$ which is the same as dividing m, $w_0 - 1$ (or $w'_0 - 1$) times, by 2. We therefore have the variable w'_0 which represents the initial word length for the RAM Model, calculated during $O(w'_0)$ time.

Therefore, storing the initial word length required $O(n + w'_0)$ time.

- 2^{w_0} We lastly modify P, the Word-RAM Model, so that the maximum input-value

length is stored in a variable *mlen* that will be used for the RAM Model to compare when `Malloc` ought to be invoked. The initialization of *mlen* takes $O(w_0)$ time because $mlen = 2^{w_0}$. We describe this process below:

1. Initialize a variable *zero* and assign it to 0.
2. Initialize a variable *mlen* and assign it to 1.
3. Initialize a variable *counter* and assign it to the value of w_0'
4. Implement a `goto` command for *counter* that jumps to step 8.
5. Implement a `multiply` command that takes the product of 2 and 2 and assigns that value to *mlen*.
6. Implement a `subtract` command that takes the difference between *counter* and 1 and assigns that value to *counter*.
7. Implement a `goto` command for *zero* that jumps to step 4.
8. Additional Code.

Therefore, the above program will run $w_0' = w_0$ times because we iterate $w_0' = w_0$ times, which is why it will take $O(w_0)$ time.

- `Operations` Addition and multiplication are redefined from the WORD RAM Model so that memory allocation is properly accounted for. All other aspects of addition and multiplication stay the same though. As such, when addition and multiplication commands are invoked such that the output is greater than *mlen*, we must cap the output to be $mlen - 1$. This additional condition can be added with $O(1)$ lines of P' because we assign the value to be $mlen - 1$ after implementing a `goto` statement checking to see if the output is greater than *mlen*.
- `Malloc` The command `Malloc` is invoked when the algorithm needs to increase its memory size beyond S'. The `Malloc` command increments S' by 1. If $S > mlen$, w_0' should also increment by 1. We implement the logic for our `Malloc` command in RAM Model below:

  1. Implement an `addition` command that adds 1 to S' and assigns that value to S'.
  2. Implement a `subtract` command that subtracts *mlen* from S' (ie. $S' - mlen$) and assigns that value to a new variable *temp*2.
  3. Implement a `goto` command for *temp*2 that jumps to step 6.
  4. Implement an `addition` command that adds 1 to w_0' and assigns that value to w_0'.
  5. We then properly update by assignment the value of *mlen* with the new value of w_0'.
  6. Additional Code

  Because there are no iterations back and forth between two `goto` commands, the above takes $O(1)$ time and therefore, when we add these additional lines for the command `Malloc` in our RAM Model, we get $O(1)$ slowdown.
- `Accessing Memory Out of Bounds` Based on the definition of the Word Ram Model given in **Definition 6.1** of class notes, if the program attempts to access memory usage beyond the maximum supported word size then there is no effect on the Word Ram model. Our RAM Model needs to accommodate for this case. More specifically, we need to check that when we `read` or `write` that the index and values that we `read` or

`write` are less than S' and $mlen$ respectively. This can be implemented by a simple `goto` statement that checks if the difference of the index and S' or the difference of the value and $mlen$ is 0 such that S' and $mlen$ are less. If it is then our program skips to the directed `goto` line and doesn't perform the operation.

- `Summary of Run Time` In summary, when we add all these additional implementations together, we get

$$\text{Time}_{P'}(x) = O\left(\text{Time}_P(x)\right) + O\left(n\right) + O\left(w_0\right) + O\left(w_0\right) + O\left(1\right) + O\left(1\right) + O\left(1\right) + O\left(1\right)$$

  where $O\left(\text{Time}_P(x)\right)$ comes from the parts of the Word RAM model that we can keep in the RAM model, $O\left(n\right) + O\left(w_0\right)$ comes from getting variable w_0, $O\left(w_0\right)$ comes from getting 2^{w_0}, $O\left(1\right)$ comes from getting S', $O\left(1\right)$ comes from additional lines of code in the addition and multiplication operators, $O\left(1\right)$ comes from the malloc command, and $O\left(1\right)$ comes from checking out of bounds indexes and values. This can all simplify down to

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

- `Conclusion` Our implementation of the RAM Program P' does not fundamentally differ from the basic operations of the Word RAM Program P. In fact, with the new implementation of the RAM Model, which simulates the Word RAM Model, we created a new command `Malloc`, added variables $mlen$, w_0', and S', implemented additional lines of code for `addition` and `multiplication` to cap memory, and ensured that there is no effect on the newly implemented RAM Model when we try to access memory outside of S' and values beyond $mlen$. All other operations we kept the same. We have therefore shown above that these additional operations take $\text{Time}_{P'}(x) = O\left(\text{Time}_P(x) + n + w_0\right)$. We have also shown that the new implementation of the RAM Program mimics completely the Word RAM Program because all operations are accounted for and perform the same functions. Thus, we can prove that P' halts on x iff P halts on x, and if they halt, then $P'(x) = P(x)$.