HW 3 - CS 120

Evan Jolley

September 2022

1

All code.

2

2.1

First RAM program:

- Lines $0-6$ will only run once each, as no *GOTO* will send us back to iterate over them again.
- Out of all lines $7-10$, 7 is the only one that is guaranteed to run. Counter is always initialized to N, and if $N = 0$, then 7 will send us straight to 11. If not, lines $7-10$ will run N times total, capped off by an additional 7 iteration when $counter = 0$, breaking the loop. Thus, line 7 runs $N + 1$ times and lines $8-10$ run N times each.
- Lines $11-12$ run once each, as once 11 is reached, no *GOTO* will send us back to iterate over them again.

$$7 * 1 + 1 * (N + 1) + 3 * N + 2 * 1$$

$$7 + N + 1 + 3N + 2$$

$$4N + 10$$

Second RAM program:

- Lines $0-7$ will only run once each, as no *GOTO* will send us back to iterate over them again.
- Out of all lines $8-14$, 8 is the only one that is guaranteed to run. Counter is always initialized to N, and if $N = 0$, then 8 will send us straight to 15. If not, lines $8-14$ will run N times total, capped off by an additional 8 iteration when $counter = 0$, breaking the loop. Thus, line 8 runs $N + 1$ times and lines $9-14$ run N times each.

- Lines $15 - 19$ run once each, as once 15 is reached, no *GOTO* will send us back to iterate over them again.

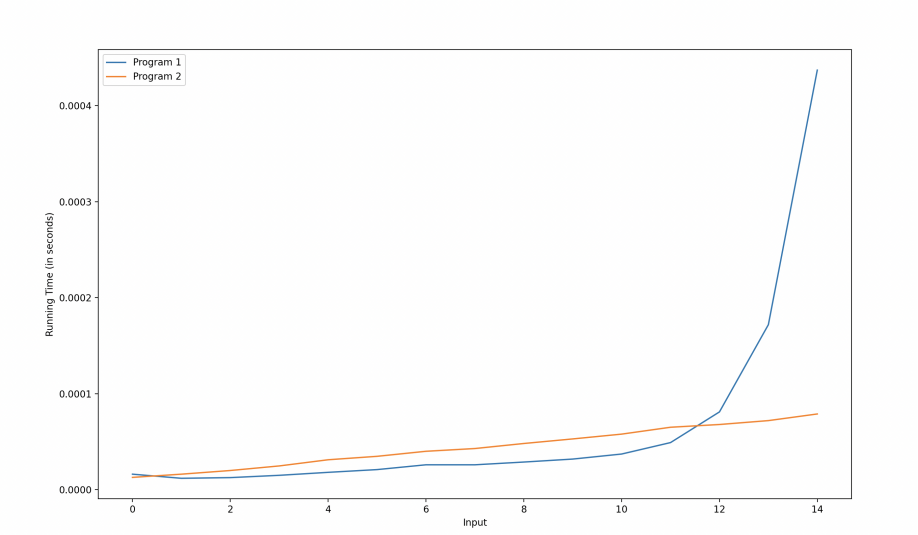
$$8 * 1 + 1 * (N + 1) + 6 * N + 5 * 1$$

$$8 + N + 1 + 6N + 5$$

$$7N + 14$$

$7N + 14 > 4N + 10$ for all N (N is defined as a natural number in the input), and thus the first RAM program is faster.

2.2



As you can see from the graph above, Program 1 is faster with input values between 0 and about 12, and Program 2 becomes faster at inputs of size > 12.

2.3

The discrepancy between $2a$ and $2b$ relates closely to something that we discussed in section. In the RAM Model, programs are able to manipulate arbitrarily large numbers, and there is no difference in the runtime between the same command run on different sizes of numbers.

This is not how Python works. At a certain size of integer, Python switches to *BIGNUM* calculations in order to deal with numbers that take more memory to store. Thus, if a Program 1 has a faster RAM runtime, but deals with larger numbers than Program 2, it might run slower in Python. Let's take a look and see if that's the case.

Our answer lies in lines $10 - 12$ of Program 2. The program takes $mod\ 2^{32}$ of $result$ during each iteration of the "GOTO loop," instead of waiting until all of 11^{2^N+1} has been calculated. Program 1 has no such reductions as it calculates 11^{2^N+1}. This means that Program 2's $result$ variable is consistently smaller than Program 1's. At large values of N, $BIGNUM$ operations are required to run Program 1, and its python code takes longer to execute commands that would theoretically take the same amount of time in the RAM Model regardless of input.

While Program 1 is faster in the RAM Model, it is slower in Python at high values of N due to the large numbers it has to manipulate necessitating $BIGNUM$ operations.

3

The Word RAM Model is similar to the RAM Model with a few extra constraints. If we're using a RAM program to mimic a Word RAM program, the RAM program is going to have to preform a few operations it wouldn't normally have to in order to make sure these constraints are satisfied.

A good analogy for this that was mentioned in Winthrop Monday OH was lists between Java and Python. In Java, the size of a list is static, but in Python it's not. If we were writing a Python program to try and emulate a Java one, we'd have to preform extra operations to make sure that our lists never changed size and add that runtime to our overall runtime. This is why we see the Word RAM runtime equal to the RAM runtime with a few extra operations. Let's discuss what these additions mean.

$$Time_{P'}(x) = O(Time_P(x) + \underline{n} + \underline{w_o})$$

Definition 6.1 in Lecture 7 notes is helpful for understanding how Word RAM differs from RAM. Let's go through each of the four bullet points defining Word RAM and discuss how we will use RAM to emulate them, considering runtimes as we go.

First Bullet

- Memory: array of length S, with entries in $0, 1, ..., 2^w - 1$. Reads and writes to memory locations larger than S have no effect.

There are a few things to unpack here. First, right off the bat, we will discuss the initialization of word length w and thus memory size S in a later bullet point.

Second, this part of Word RAM clearly differs from basic RAM, as in RAM we have arbitrarily large memory. In order to prevent our RAM from accessing memory it should not, we will need to use $IF...GOTO$ statements. For example,

the $IF...GOTO$ command could skip over read and write lines attempting to access entries in memory greater than $2^w - 1$. $IF...GOTO$ statements run in constant time $O(1)$ in both RAM and Word RAM.

Given the constant runtime of this bullet point, our added runtimes do not come from this part of Word RAM's definition.

Second Bullet

- Operations: Addition and multiplication are redefined from RAM Model to return $2^w - 1$ if the result would be $\geq 2^w$

This part of Word RAM can be handled in a similar way as our first example. If we want to reassign all sums and products to be $2^w - 1$ if the result would be $\geq 2^w$, then we can use simple $IF...GOTO$ statements to determine what assign command is run (e.g., assign to the true product or assign to $2^w - 1$). $IF...GOTO$ statements run in constant time $O(1)$ in both RAM and Word RAM.

Again, given the constant runtime of this bullet point, our added runtimes do not come from this part of Word RAM's definition.

Third Bullet

- Initial settings: When a computation is started on an input x, which is an array consisting of n natural numbers, the memory size is taken to be $S = n$, and word length is taken to be $w = \lfloor log(max(S, x[0], ..., x[n-1])) \rfloor + 1$.

The part of note in this bullet point is the initialization of w through the expression:

$$w = \lfloor log(max(S, x[0], ..., x[n-1])) \rfloor + 1.$$

First, finding the max of $n+1$ different values (in this case $(S, x[0], ..., x[n-1])$) will run in time $O(n+1) = O(n)$, as we will need to traverse the entire set, using $IF...GOTO$ statements to find the greatest value.

Second, taking the log will require us to divide iteratively by the base, as this operation is not defined in either RAM or Word RAM. We know that inputs and S are bound by 2^w, so we will need to divide the max by 2 at most w times to find log_2. Thus, this runs in time $O(w_o)$, where w_o is our initial word size.

Finally, the floor function and adding 1 both run in constant time and function the same in RAM and Word RAM. They will not contribute to our additions to total runtime.

After digging into this part of our definition, we now know we must add $O(n)$ and $O(w_o)$ to RAM's runtime in order to emulate Word RAM.

Fourth Bullet

- Increasing S and w: If the algorithm needs to increase its memory size beyond S, it can issue a $MALLOC$ command, which increments S by 1 and if $S = 2^w$, it also increments w.

$MALLOC$ increments S by a constant amount. Using an $IF...GOTO$ statement to check if $S = 2^w$, it might also increment w by a constant amount. All three of these operations run in constant time! This bullet point will not account for our runtime additions either.

Conclusion

As outlined, the third bullet point accounts for the addition of n and w_o to the RAM Model's runtime in our attempt to use it to simulate Word RAM. The three other parts of Word RAM's definition contribute constant amounts of runtime which will be dominated by n and w_o.

$$Time_{P'}(x) = O(Time_P(x) + \underline{n} + \underline{w_o})$$