CS 120: Intro to Algorithms and their Limitations

Problem Set 3

Due: October 4, 2023 11:59pm **Denny Cao**

Collaborators:
No. of late days used on previous psets: 0
No. of late days used after including this pset: 1

§2 Empirically Evaluating Simulation Runtimes and Explaining Them Theoretically

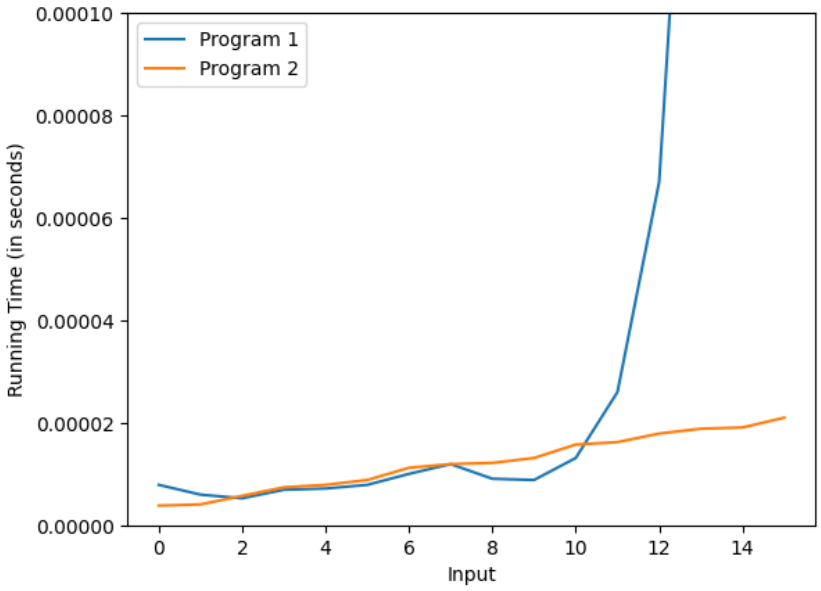
(a) Let $P_1(x)$ and $P_2(x)$ denote the outputs of the first and second program respectively given an input x. Then, $\text{Time}_{P_1}(N) = 10 + 4N$ and $\text{Time}_{P_2}(N) = 14 + 7N$.

We obtain this by observing that the first program executes 7 commands from lines 0 to 6 before entering a loop that executes 4 commands from lines 7 to 10 N times. The last time the loop runs, it will go back to line 7 which will exit the loop to line 11, executing another command. The program then executes 2 more commands from lines 11 to 12, giving us a total of $\text{Time}_{P_1}(N) = 7+4N+1+2 = 10+4N$.

For the second program, there are 8 commands before entering a loop that executes 7 commands from lines 8 to 14 N times. The last time the loop runs, it will go back to line 8 which will exit the loop to line 15, executing another command. The program then executes 5 more commands from lines 15 to 19, giving us a total of $\text{Time}_{P_2}(N) = 8 + 7N + 1 + 5 = 14 + 7N$.

As $10 + 4N < 14 + 7N$, Program 1 runs faster for all $N \in \mathbb{N}$.

(b) Program 2 is faster. However, for smaller inputs than 12, Program 1 is faster. At $N \geq 12$, Program 2 is faster.



(c) The discrepancies we see between Parts 2a and 2b are due to the assumption in the RAM model that every operation takes $O(1)$ time. However, in reality, as the numbers in the programs grow large rapidly, the time it takes for each operation will depend on the size of the input. This is only an issue in Program 1, as the value of `result` is not bounded and will grow exponentially large and will quickly exceed the limit that can be stored in a word, requiring multiple words to store the value of `result`. This creates the exponential curve we see in (2b) rather than the linear curve we expect to see in (2a). In contrast, Program 2 bounds the value of `result` by taking `result` mod 2^{32} in every iteration of the loop, allowing `result` to be stored in one word, and thus the operations on `result` for Program 2 can be $O(1)$ which is why Program 2 is linear both in (2a) and (2b).

(d) We can bound the runtime by observing the true runtime of the multiplication operator. In Python, when multiplying large numbers, Karatsuba multiplication is used, which runs in $O(N^{\log_2 3})$ where N is the maximum number of bits of the factors.

For Program 2, the maximum number that is multiplied is 2^{32}. Thus, multiplication would run in $O\Big(\big(2^{32}\big)^{\log_2 3}\Big) = O\Big(\big(2^{\log_2 3}\big)^{32}\Big) = O(3^{32}) = O(1)$ time, and thus multiplication takes constant time, resulting in the linear curve we see in (b), as multiplication is computed n times, resulting in $O(n)$.

For Program 1, the largest multiplication executed is $17^{2^n} \times 17$. Thus, we can bound the runtime of Program 1 by overestimating the runtime for multiplication as the runtime it takes to execute $17^{2^n} \times 17$. The maximum number of bits of the factors is $\log_2 17^{2^n} = 2^n(\log_2 17)$. Thus, with Karatsuba multiplication, the runtime will be $O\Big((2^n \log_2 17)^{\log_2 3}\Big) = O\Big((2^n)^{\log_2 3}(\log_2 17)^{\log_2 3}\Big) = O\Big(3^n(\log_2 17)^{\log_2 3}\Big)$. As $(\log_2 17)^{\log_2 3}$ is a constant, $O\Big(3^n(\log_2 17)^{\log_2 3}\Big) = O(3^n)$. As multiplication is computed n times, the runtime of Program 2 can be bounded by $O(n3^n)$, creating the exponential curve we see in (b).

§3 Simulating Word-RAM by RAM

Proof. We will take a simulation strategy by using an implementation-level description.

Computability (Operations):

(i) **Initialization**: In P', compute variables `word_len` and `mem_size`.

As `mem_size` is the length of the input, we set `mem_size` = `input_len`. Then, `word_len` can be computed as follows:

$$\texttt{word\_len} = \lfloor \log \max\{\texttt{mem\_size}, x[0], \ldots, x[n-1]\} \rfloor + 1$$

We can break computing `word_len` into two parts. For the first part, we find the largest element of the n inputs. We can implement this in a RAM Program by iterating over the n inputs and creating a variable `max` that stores the current maximum element, making comparisons between the current input element and `max`, setting `max` to the current input element if the element is larger than `max`. At

the end of the loop, `max` will be the largest element of the n input elements and can then be compared to `mem_size` to see which is larger. Let this value be k.

For the second part, the length of k can be found by computing $\lfloor \log_2 k \rfloor$, which we can implement in a RAM Program by continually setting $k = \lfloor \frac{k}{2} \rfloor$ until $k = 0$ and counting how many divisions are needed. We then add 1 to this value to obtain `word_len`.

For convenience, we compute $\texttt{max\_num} = 2^{\texttt{word\_len}}$. We implement this in a RAM Program by setting `max_num` to 1 and then setting a variable `temp` to `word_len`. Then, multiply `max_num` by 2 and then decrement `temp` by 1 and then repeat until `temp==0`.

(ii) **Memory**: In P', when reading and writing to memory locations larger than `mem_size`, nothing occurs. We can implement this in a RAM Program by adding an additional check to the read and write operations to check if, for the memory location $M[i]$, if $i \leq \texttt{mem\_size}$, and if it is, then continue with the operation. By doing so, if the memory location is greater than `mem_size`, nothing will occur.

(iii) **Operations**: All operations are the same as P, but the addition and multiplication operations are redefined to return $2^{w_0} - 1$ if the result is $\geq 2^{w_0}$. We can implement this in a RAM Program by checking if the result of the addition and multiplication operation is greater than `max_num`. Let $\texttt{var}_i = \texttt{var}_j + \texttt{var}_k$ or $\texttt{var}_i = \texttt{var}_j \times \texttt{var}_k$. Then, after computing the operation, check if $\texttt{var}_i \geq \texttt{max\_num}$, and if it is, set $\texttt{var}_i = \texttt{max\_num} - 1$.

(iv) **MALLOC**: In P', `MALLOC` is an operation that increases memory size beyond `mem_size`. We can implement this in a RAM Program by incrementing `mem_size` by 1, setting $M[\texttt{mem\_size} - 1] = 0$, and if $\texttt{mem\_size} = 2^{\texttt{word\_len}}$, it also increments `word_len` by 1. We also double `max_num` by 2 in order to maintain the fact that $\texttt{max\_num} = 2^{\texttt{word\_len}}$.

Runtime (Operations):

(i) **Initialization**: When computing `word_len`, the first part we discussed in the operation is done by iterating through the n input elements and drawing a comparison between the element and the `max` variable, taking $O(n)$ time. For the second part, if k can be represented in w_0 bits, then $k \leq 2^{w_0} - 1$. Thus, $\lfloor \log_2 k \rfloor \leq w_0 - 1$, meaning that we will have to do at most w_0 divisions before exiting the loop. As all other operations take $O(1)$ time in our implementation of the second part, it will take $O(w_0)$ time to compute the second part. Thus, in total, to compute `word_len`, it takes $O(n + w_0)$ time. When setting $\texttt{mem\_size} = \texttt{input\_len}$, reading from memory takes $O(1)$ time. Computing `max_num` is done through a loop that computes basic operations `word_len` times, and thus takes $O(w_0)$ time. Thus, the initialization of P' can be bounded by $O(n + w_0)$ time.

(ii) **Memory**: For read and write operations, an additional check is needed to check if the memory location is less than or equal to `mem_size`, which will take $O(1)$ time.

(iii) **Operations**: For addition and multiplication operations, an additional check is needed at the end of the operation to check if the result is greater than or equal to `max_num` which takes $O(1)$ time and if it is, then set it to $\texttt{max\_num} - 1$, which takes $O(1)$ time.

(iv) **MALLOC**: Incrementing, setting, and multiplying are all basic operations and thus can be done in $O(1)$ time.

Computability (Program): For every Word-RAM program P, there is an ordinary RAM program P' that performs the same operations, except it replaces every operation specified in **Computability (Operations)** with the corresponding substitute. We have shown that our simulated Word-RAM program is equivalent to the original Word-RAM program, as we have shown substitutes for the differences and kept all other operations the same. Thus, for every Word-RAM program P, there exists a modified RAM program P' that will have the same output as P on every input x. Thus, if P' halts on $x \iff P$ halts on x, and if they halt, then $P(x) = P'(x)$, and the first part of the theorem is proven.

Runtime (Program): Every change we made to an operation took a constant number of steps, and thus is $O(1)$. The initialization step takes $O(n + w_0)$ time, and thus our overall time is:

$$\begin{aligned} T_{P'}(x) &= O(1)(T_P(x)) + O(n + w_0) \\ &= O(T_P(x) + n + w_0) \end{aligned}$$

and the second part of the theorem is proven. □