CS 203 Midterm

Alex Moening
10/22/2023

A. The algorithm computes the sum of a number's digits, after each digit is raised to the power of the count of digits. It does this by first counting the number of digits using a counter and integer division to move through each digit of a copy of the input number until the copy reaches 0. It then iterates through another copy of the number, using modulus to look at the last digit and multiply it by itself using the digit count found previously. It then adds that number to a holding variable, integer divides the copy number and repeats for every digit until the copy number is 0. Lastly, it compares the holding variable value (the sum of the digits multiplied by themselves) by the original value to see if they are the same (returns true) or not (returns false)

B. The input the function is dependent on is the number of digits of the input number (n = number of digits). It is not necessary to analyze a best and worst case since the code will perform the exact same functions given all inputs within the range of a particular size (for example, it will run the same amount of loops when given 3 vs 6, and the same holds true for 153 vs 958). As such, we can predict the runtime based on the counters of the functions that change with the input (digits) and create a general count of the basic operations.

C. There are 3 basic operations:
   a. Operation 1; the compare function contained in the while loop check on line 3, counts how many times the while loop and its interior functions occur
   b. Operation 2; the compare function contained in the while loop check on line 8, counts how many times the second while loop performs, including occurrences where the interior if condition fails
   c. Operation 3 is the multiplication function inside the for loop on line 12, counts how many times the for loop inside the if condition of the second while loop occurs

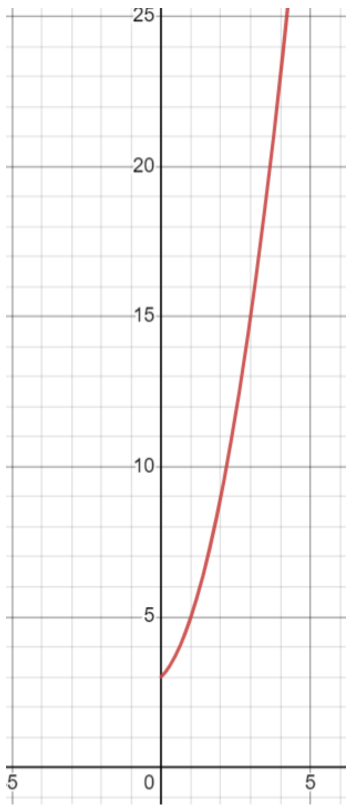
D. Basic Operations as Summations
   a. $C_1(n) = \sum_{i=1}^{n+1} (1)$
   b. $C_2(n) = \sum_{i=1}^{n+1} (1 + C_3(n)) = \sum_{i=1}^{n+1} (1 + \sum_{j=0}^{n} (1))$
   c. $C_3(n) = \sum_{j=0}^{n} (1)$
   d. $C(n) = C_1(n) + C_2(n) = \sum_{i=1}^{n+1} (1) + \sum_{i=1}^{n+1} (1 + \sum_{j=0}^{n} (1))$

E. Basic Operation count as closed form equations
   a. $C(n) = \sum_{i=1}^{n+1} (1) + \sum_{i=1}^{n+1} (1 + \sum_{j=0}^{n} (1))$

$$= \sum_{i=1}^{n+1} (1) + \sum_{i=1}^{n+1} (1 + n)$$
$$= \sum_{i=1}^{n+1} (1) + \sum_{i=1}^{n+1} (1) + \sum_{i=1}^{n+1} (n)$$
$$= (n + 1) + (1)(n + 1) + (n)(n + 1)$$
$$= (n + 1) + (n + 1) + (n^2 + n)$$
$$= n^2 + 3n + 2$$

F. Graph Shown on Right

a. The graph shows how the basic operation count grows with n^2 as the primary growth factor. This is a standard function of growth that is commonly seen. It is a quadratic growth factor, and as such, is reasonable for implementation and usage even with larger inputs. However, it would not be the most practical for processing large datasets of inputs since it is above a linear complexity range, becoming much more computationally expensive than a linear or logarithmic type growth factor.



(x axis, input (num degrees); y axis C(n)

G. In terms of time-efficiency, the code will perform quickly with small digit numbers, but will easily become computationally more expensive and require increasingly more run-time as the input grows larger, until it reaches the integer input size cap. The function is of quadratic efficiency form, as a result of the for loop nested in the while loop. It is considerably more efficient than exponential or factorial growth factors, but not quite as efficient as linear, constant, or logarithmic growth factors. The count equation is representative of the average case, and can be written C(n) ∈ Θ(n) in asymptotic notation.