K. Hohmeier

CSC 380

Homework 2 Written Report

Due September 15, 2020, 11:59 PM

Problem 1: Analyzing the Algorithm "Mystery"

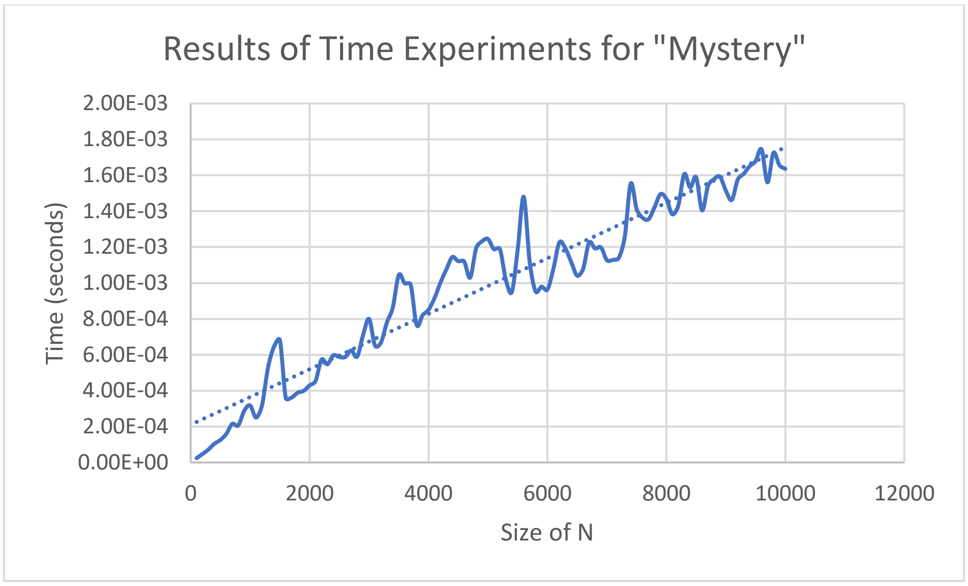
For this problem, the algorithm "Mystery," as described in Problem 4 in Exercise 2.3 from the textbook, was analyzed and explored. Aside from analyzing the pseudocode provided in the textbook, the algorithm was also implemented in Python.

a. **What does this algorithm compute?** The algorithm finds the sums of squares up to the first n natural numbers. That is, it computes the sum of the squares of each consecutive integer in the range from 1 to a specified number n.

b. **What is its basic operation?** The basic operation of this algorithm is the addition and the multiplication through each iteration of the for loop. The multiplication dominates the addition.

c. **How many times is the basic operation executed?** This operation is executed n times, depending on what the value of n is, because of the for loop.

d. **What is the efficiency class of this algorithm?** This algorithm appears to be O(n), from the above analysis. Timing experiments will be used to determination the accuracy of this conclusion, the results of which are shown in Graph 1 below. These experiments confirm that the algorithm does appear to be O(n).



Graph 1. Results of time experiments for the algorithm "Mystery."

e. **Suggest an improvement, or a better algorithm altogether, and indicate its efficiency class. If you cannot do it, try to prove that, in fact, it cannot be done.** Computing the sum of the squares of the first n natural numbers has a simple formula: $\frac{n(n+1)(2n+1)}{6}$. This fact can proved via induction. It would be much more efficient to implement the algorithm using this formula, since this can be implemented in O(1) efficiency.

Problem 2: Analyzing the Algorithm "Enigma"

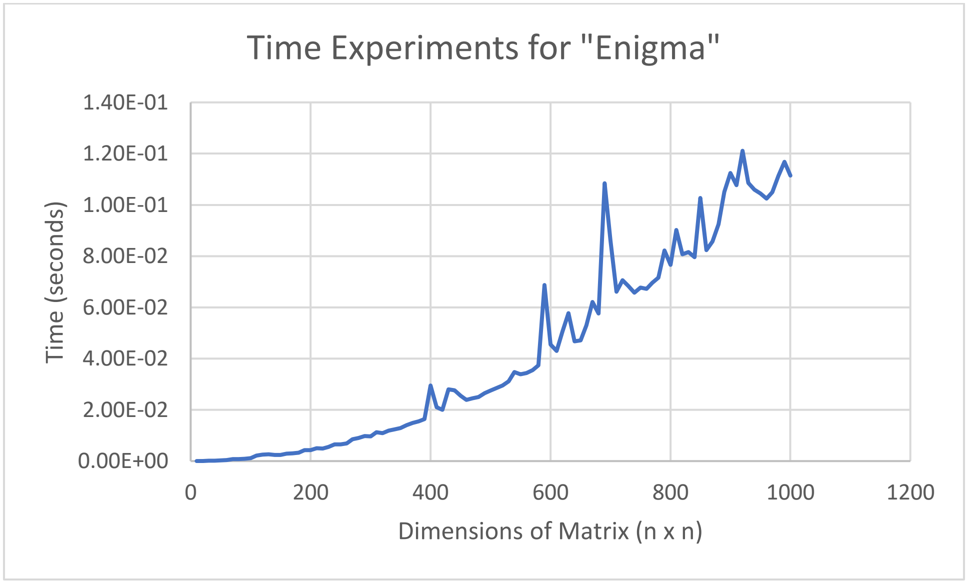
For this problem, the algorithm "Enigma," as described in Problem 6 in Exercise 2.3 from the textbook, was analyzed and explored. Aside from analyzing the pseudocode provided in the textbook, the algorithm was also implemented in Python.

a. **What does this algorithm compute?** The algorithm determines whether the square matrix is symmetric or not. It returns "true" if the matrix is symmetric and "false" otherwise.

b. **What is its basic operation?** The basic operation is the comparison of the matrix elements.

c. **How many times is the basic operation executed?** The number of comparisons performed will depend on whether we are dealing with best-case, worst-case, or average case. Best case will occur when the first comparison shows that the matrix is not symmetric, which is O(1) efficiency. In worst case, there will be $\frac{n^2}{2}$ comparisons performed. In average case, there will be $\frac{1}{n}\sum_{i=0}^{n-1} i = \frac{(n-1)n}{2n} = \frac{n-1}{2}$ comparisons.

d. **What is the efficiency class of this algorithm?** Based on the worst-case performance, this algorithm appears to be O(n^2). Timing experiments will be used to determination the accuracy of this conclusion, the results of which are shown in Graph 2 below. These results confirm that the algorithm does appear to be O(n^2).



Graph 2

e. **Suggest an improvement, or a better algorithm altogether, and indicate its efficiency class. If you cannot do it, try to prove that, in fact, it cannot be done.** There is not a more efficient way to implement this matrix operation, since there is no way to determine whether the matrix is symmetric or not without looking at every element of the matrix.

$O(n^2)$ is most likely a lower bound for the efficiency of checking whether a matrix is symmetric or not.

Problem 3: Analyzing the Algorithm Q(n)

For this problem, the algorithm Q(n), as described in Problem 4 in Exercises 2.4 from the textbook, was analyzed and explored. Aside from analyzing the pseudocode provided in the textbook, the algorithm was also implemented in Python.

a. **Set up a recurrence relation for this function's values and solve it to determine what this algorithm computes.** The recurrence relation can be obtained from the algorithm: $Q(n) = Q(n-1) + 2n - 1$, with initial condition $Q(1) = 1$.

   To solve this recurrence relation, we can insert different values of n to see if we can observe a pattern.

   $Q(2) = Q(1) + 4 - 1 = 4$

   $Q(3) = Q(2) + 6 - 1 = 9$

   $Q(4) = Q(3) + 8 - 1 = 16$

   …

   $Q(n) = n^2$

   Hence, we can see that the algorithm computes the square of n. More values of n can be tested by running a for loop in Python to verify that this algorithm does indeed compute the square of n.

b. **Set up a recurrence relation for the number of multiplications made by this algorithm and solve it.** Denote the number of multiplications by M(n). If the "if" statement is true, then no multiplications are performed. If the "if" statement is false, then 1 multiplication is performed. Hence, the recurrence relation is $M(n) = M(n-1) + 1, M(1) = 1$. To solve this recurrence, we can examine different values of n to identify a pattern:

$M(1) = 1,\ M(2) = M(1) + 1 = 2, M(3) = M(2) + 1 = 3, M(4) = M(3) + 1 =$
$4, \dots M(n) = n$

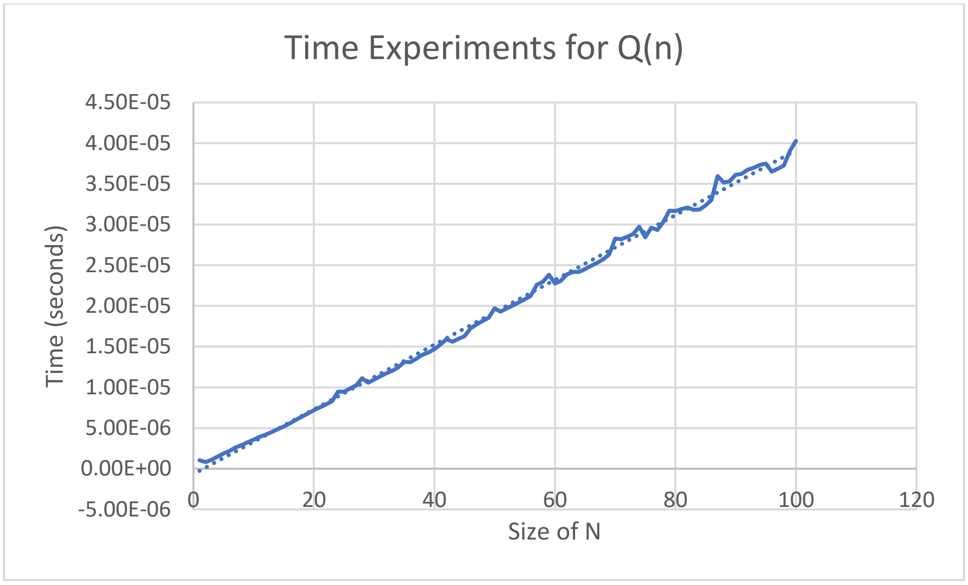
Hence, we see that the recurrence relation, when solved, is $M(n) = n$.

c. **Set up a recurrence relation for the number of additions/subtractions made by this algorithm and solve it.** Denote the number of additions and subtractions by A(n). If the "if" statement is true, then no additions or subtractions are performed. If the "if" statement is false, then 1 addition and 1 subtraction will be performed. Hence, the recurrence relation is $A(n) = A(n-1) + 2, A(1) = 1$. To solve this relation, we can test different values of n to see if there is a pattern.

$C(1) = 0, C(2) = C(1) + 2 = 2, C(3) = C(2) + 2 = 4, C(4) = C(3) + 2 =$
$6, \dots C(n) = 2n - 2$

Hence, we see that the recurrence relation, when solved, is $C(n) = 2n - 2$.

d. **Write a program to experimentally verify that the big-O of this algorithm is consistent with your results from part b and part c.** Based on the results of the analysis of the recurrence relation in parts b. and c., it is hypothesized that the algorithm is O(n). Timing experiments will be performed to see if this conclusion is correct. The results of these experiments are shown in Graph 3 below. Note that the number of experiments that could be run was limited to the maximum recursion depth in Python. The results of these experiments indicate that the algorithm is O(n), as expected, although there are a few "bumps" in the graph.



Graph 3. Results of the time experiments for the algorithm Q(n)

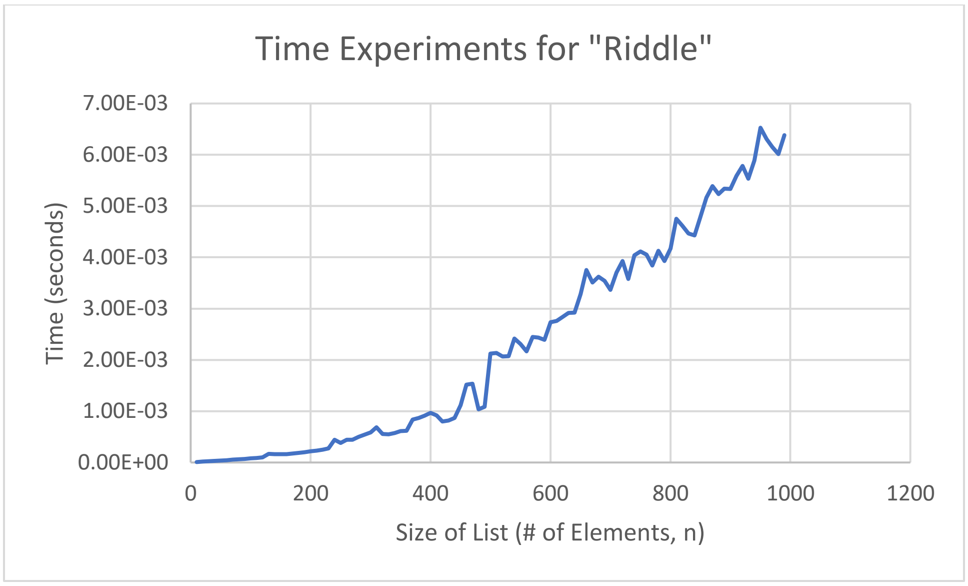
Problem 4: Analyzing the Algorithm Riddle(n)

For this problem, the algorithm Riddle(n), as described in Problem 9 in Exercises 2.4 from the textbook, was analyzed and explored. Aside from analyzing the pseudocode provided in the textbook, the algorithm was also implemented in Python.

a. **What does this algorithm compute?** The algorithm finds the smallest element in a list.

b. **Set up a recurrence relation for the algorithm's basic operation count and solve it.** Let $R(n)$ denote the number of basic operations this algorithm utilizes. First, we note that the basic operation is

c. **Write a program to experimentally the big-O of this algorithm is consistent with your results from part b.** We anticipate that the big-O of this algorithm will be O(n^2). Timing experiments will show whether this hypothesis is correct or not. The results of these experiments are shown in the graph below (Graph 4). The graph indicates that the algorithm is O(n^2).



Graph 4. Results of the time experiments for the algorithm Riddle