K. Hohmeier

CSC 380

Homework 3 Written Report

Due September 24, 2020, 11:59 PM

Problem 1: Implement SelectionSort and BubbleSort

For this problem, the selection sort and bubble sort algorithms, as described in chapter 3 of the class textbook, were implemented in Python. In addition, timing experiments were performed for both algorithms. The results of these experiments can be seen in Figure 1 below.

We observe that bubble sort generally performs worse than selection sort, although both are still ultimately O(n^2) algorithms. Selection sort and bubble sort perform the identical number of swaps, $\frac{n(n-1)}{2}$, hence giving both algorithms O(n^2) efficiency, but selection sort performs $n-1$ swaps, while bubble sort performs anywhere from 0 (in best case) to $\frac{n(n-1)}{2}$ (in worst case). This means that selection sort will generally run slightly faster than bubble sort, since its swapping is O(n) efficiency, while bubble sort's swapping is O(n^2) efficiency.

To determine the degree to which selection sort outperformed bubble sort, the ratio of each bubble sort time to each selection sort time was calculated, and then an average was taken of those values. These calculations showed that bubble sort took roughly 17% longer than selection sort to run.

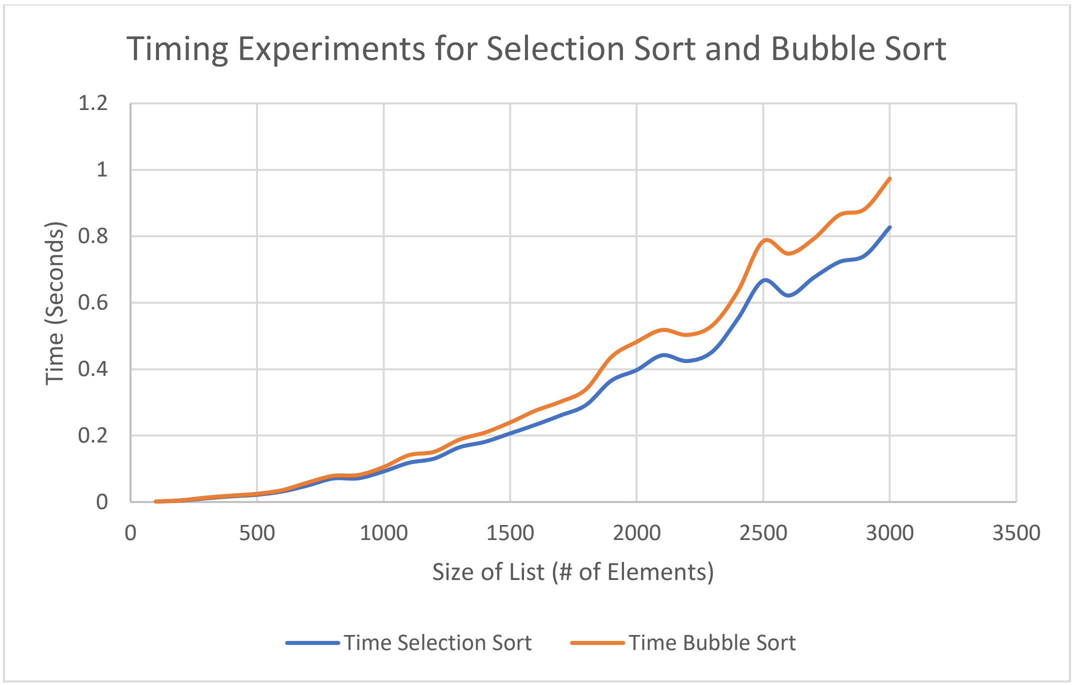

Figure 1. Graph of times for bubble sort and selection sort

Problem 2: Implement BruteForceStringMatch and Count Comparisons

For this problem, we will first calculate by hand the number of comparisons, both successful and unsuccessful, that the brute force string matching algorithm will make for three different binary strings of length five in a binary text of one thousand zeros.

a. 00001

With this binary string, we will have a worst-case situation when applying the algorithm. Hence, the number of comparisons for this five-digit string will be 4,980, by applying the formula given in the text of m(n – m + 1), where m = the length of the pattern we are looking for and n = the length of the text being checked.

b. 10000

The number of comparisons for this five-digit string will be 996. Because the first digit in this string is a 1, it will always fail each comparison on the first digit. Hence, the 1 will be compared and then shifted down, up until the 996th digit. The 996th digit will be the final comparison, and so there are 996 total comparisons performed.

c. 01010

The number of comparisons for this five-digit string will be 1992. When the algorithm operates on this string, it will shift for every second number. That is, it will match the 0, but then fail to match the 1, and so it will shift the binary digit pattern. This will be performed until the 996th digit. Hence, there are 996 * 2 = 1992 total comparisons made.

After performing these calculations by hand, the brute force string matching algorithm was implemented in Python, and character comparisons counts were calculated in the program and compared with the earlier computations. The number of comparisons obtained by counting in the Python program was the same as the numbers obtained via the hand calculations performed above. All of the results are shown in Table 1.

String	# Comparisons (Hand Calculations)	# Comparisons (Python Calculations)
00001	4,980	4,980
10000	996	996
01010	1992	1992

Table 1. Number of Comparisons Performed by Brute Force Algorithm, determined by hand and in Python

Problem 3: Implement BruteForceClosestPair

For this problem, two versions of the brute force algorithm for determining the two closest pairs of points were implemented. The first included a square root calculation in the nested loop, which provided the actual distance between the two closest points. The second version of the algorithm skipped the square root step, which still determined the two closest points, but did not provide the correct distance. The purpose of these two implementations was to compare their efficiency in timing experiments to see how each performed.

To obtain a list of points, a Point class was created, and a list of random points were created via a helper function. These random lists of points were then used to compare the two brute force algorithms. The helper function also used huge integer values for the points in order to maximize the timing differences between the two algorithm implementations. In addition, the brute force algorithm was slightly modified so that it would return not only the distance between the two closest points but also a list containing the two closest pairs of points. After performing the timing experiments for the two algorithms, the results were graphed (see Figure 2). Although both algorithms are clearly O(n^2) efficiency, the algorithm implementation without the square root ran slightly faster than the algorithm that utilized the square root to find the correct distance between the points.

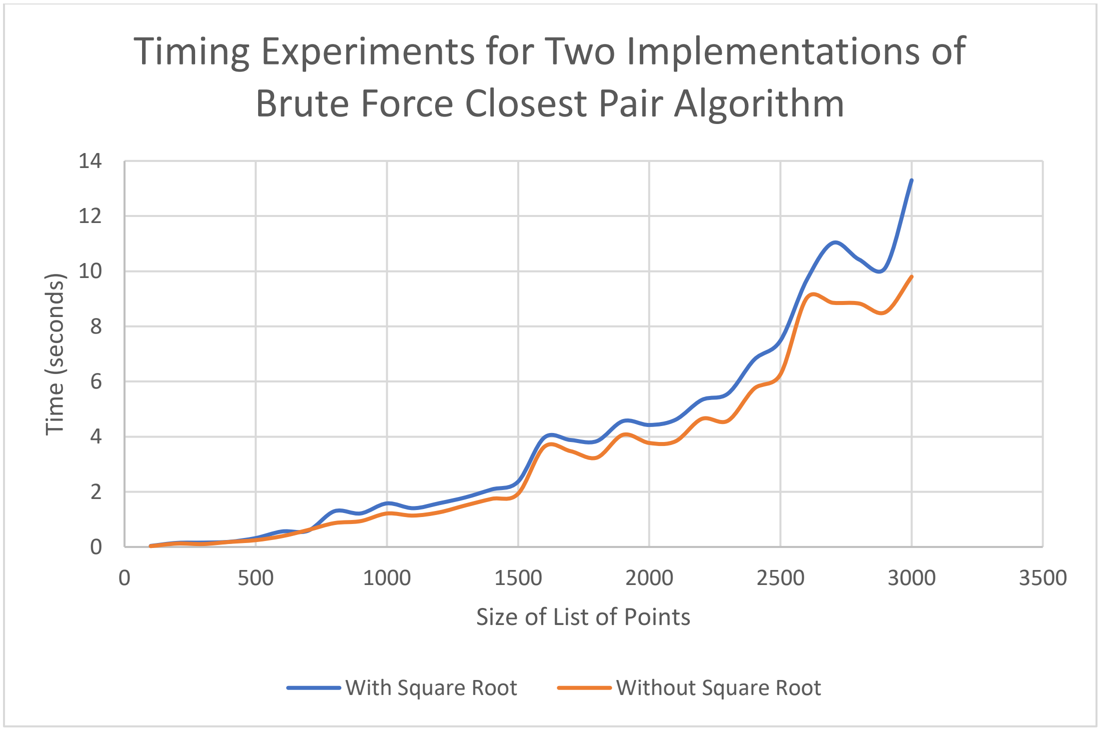

Figure 2. Graph of timing experiments for each implementation of Brute Force Closest Pair

Problem 4: Eight-Queens Chess Problem

For this problem, we consider the eight-queens chess problems, which involves placing eight queens on an 8 x 8 chessboard so that no two queens are in the same row or in the same column or on the same diagonal.

a. **How many different positions are there so that no two queens are on the same square?** This would represent the total number of ways to place each queen on the board. This is found by first considering the total number of squares on the board (64), the number of queens (8), and the number of blank positions (56). Because the queens are the same, we can permute the queens around the board and achieve the same solution. Hence, the total number of positions so that no two queens are on the same square is $\frac{64!}{56!8!} = 4{,}426{,}165{,}368$ ways.

b. **How many different positions are there so that no two queens are in the same row?** Since there are 8 possible locations for each queen in each row, there are $8^8 = 16{,}777{,}216$ possible positions to place each queen so that no two queens are in the same row.

c. **How many different positions are there so that no two queens are in the same row or in the same column?** Each time we place a queen, we remove that row and column as a possible location for the next queen. For example, when we place the first queen, we remove that row and column from consideration for placement of the next queen, leaving us with a 7 x 7 chessboard, and so forth. This yields $\frac{64*49*36*25*16*9*4*1}{8!} = 40{,}320$ positions where no two queens are in the same row or in the same column.

d. **Also estimate how long it would take to find all the solutions to the problem by exhaustive search based on each of these approaches on a computer capable of checking 10 billion positions per second.**

$$\frac{4{,}426{,}165{,}368}{10^{10}} = 0.4426\ seconds$$

$$\frac{16{,}777{,}216}{10^{10}} = 0.0017\ seconds$$

$$\frac{40{,}320}{10^{10}} = 4.032\ x\ 10^{-6}\ seconds$$