Kwasi Osae-Kwapong HW2 3/26/21

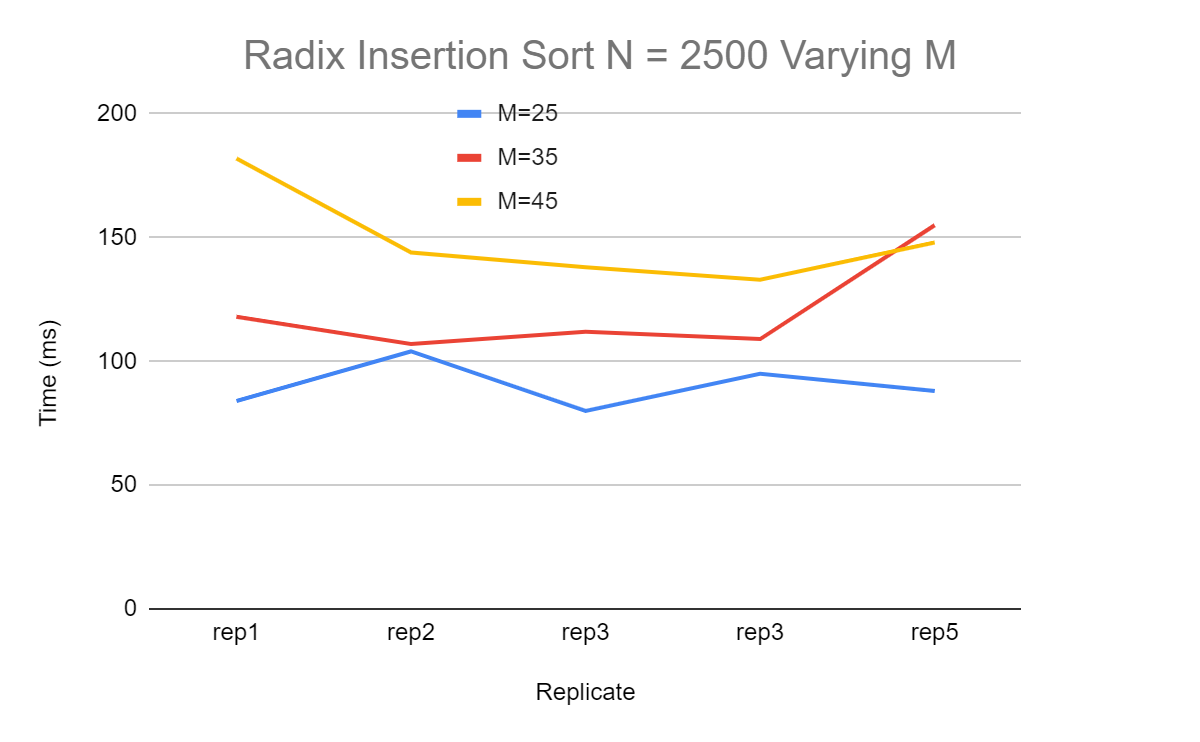
Here the running time of two Radix sort functions, one using an insertion sort implementation and the other using a counting sort implementation,are compared when sorting an array of length N of random strings of length M. Both secondary sorting algorithms have been modified to sort based on a string’s dth character and if the string does not contain a character at the dth position it will be interpreted as zero for the sorting.

Radix sort is a 1900s erra sorting algorithm developed by IBM and is a non-comparison sorting algorithm that depends on a secondary stable sorting algorithm to sort its input. Radix sort works by sorting each column or place of a digit one at a time. For example an input of integers would have all the ones, tens, and hundredths place sorted one at a time to create a fully sorted array. To achieve this, the algorithm starts from the right-most position of the random string, loops over the m-parameter going down to zero and for each iteration we call the secondary stable sorting algorithm, either the counting sort or insertion sort.

A working insertion sort algorithm that sorts by the dth element of a random string first ensures that the random string actually contains a dth element and if it does not we add the integer zero in dth position. This is necessary so that there are no empty characters when the input, which consists of one letter from each of the elements in the array A, is sorted. The strings are then sorted lexicographically. This implementation of insertion sort preservese the runtime as no additional loops are added and the while-loop maintains the two comparisons, and therefore, the algorithm is expected to run at 𝚯 n2 .This algorithm was then tested with varying input array size, N, and random string sizes, M, over five iterations per condition.

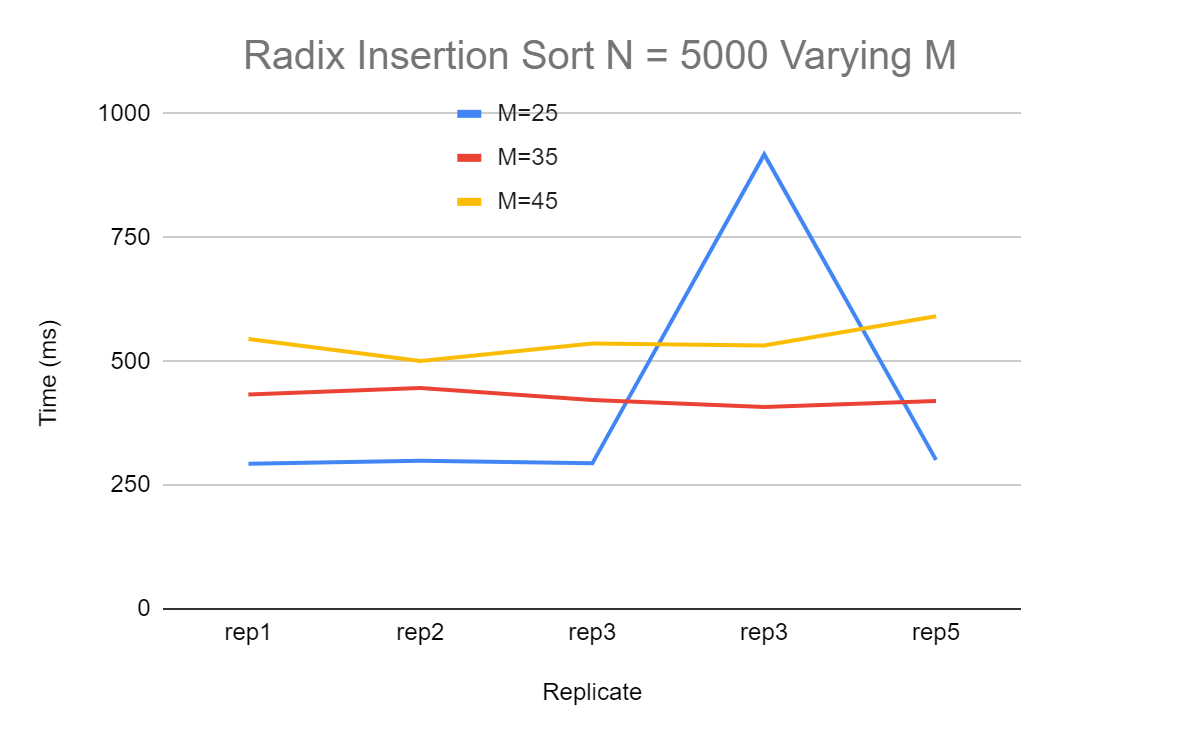
In Figure 1 we compare the radix insertion sort with an N value of 2500 and varying M values of 25, 35, and 45. As expected, the runtime increases significantly with increasing random string length M, with an M = 45 taking an average sorting time of 149 ms compared to 90 ms when M = 25.

*Figure 1: Radix Insertion Sort N = 2500 Varying M values*

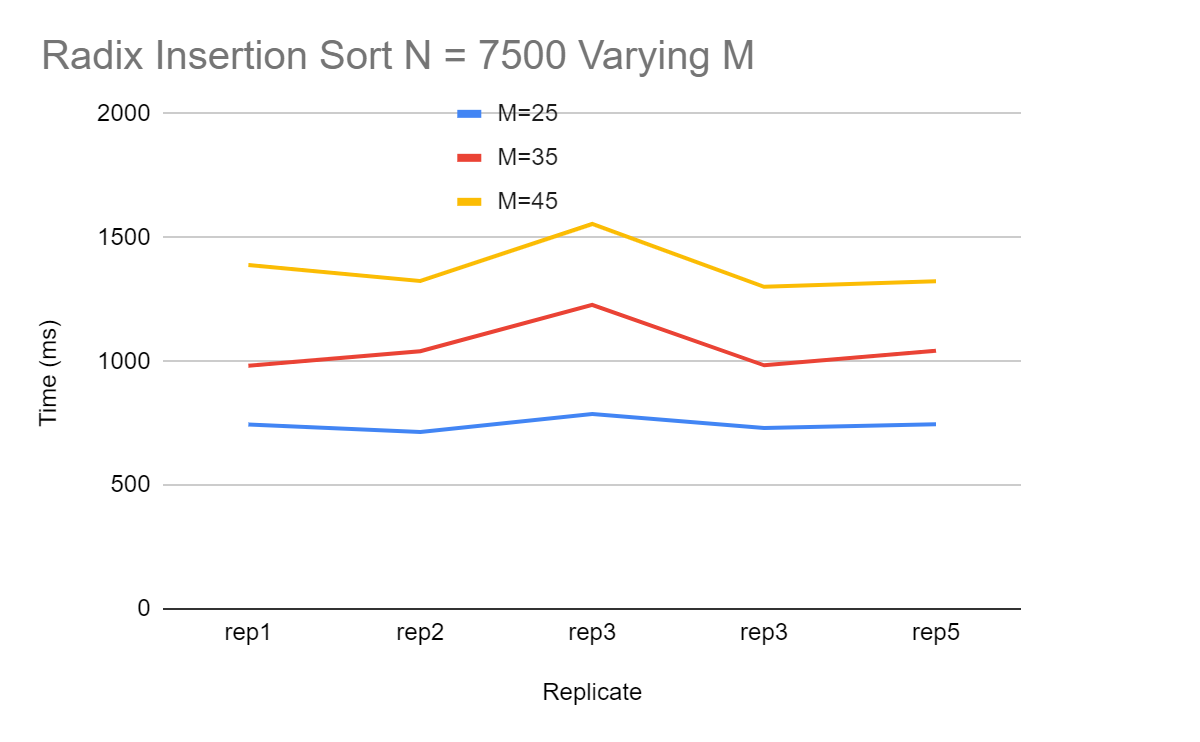


Figures 2, 3, and 4 Show a similar trend of increasing runtime with increasing M values with the exception of an outlier in Figure 2. One iteration of the M = 25 condition resulted in a runtime of nearly 900 ms, which is significantly slower than all other replicates on the chart. This may be an interesting case of the worst case scenario insertion sort where the input is reverse sorted. The trend is seen most clearly in Figure 3 as none of the plotted lines overlap. In Figure 4 the trend continues, however, the last replicate of M = 35 runs just as slow as the last replicate of M =45.

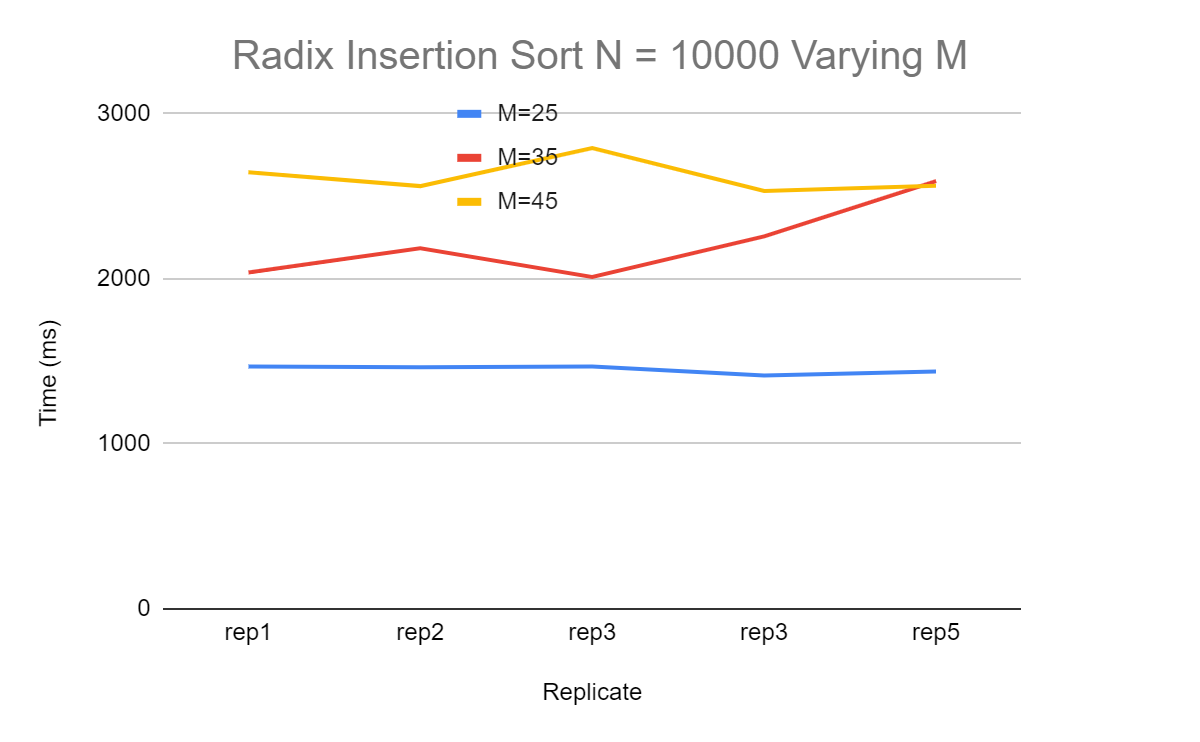
*Figure 2: Radix Insertion sort N = 5000 Varying M values*



*Figure 3: Radix Insertion Sort N = 7500 Varying M*



*Figure 4: Radix Insertion Sort N =10000 Varying M Values*



Similar to Radix Insertion sort, the Radix Counting sort input is one column of all the random string elements. Counting sort starts by initializing the auxiliary integer array C with a length of n or a maximum value of 256, and then all values in array C are set to zero. Following this the length of the random string is checked for the dth element, and if the dth element exists the dth index of auxiliary array C is incremented. This step leads to the array C containing integers that correspond to how many of a given dth element there are in array A. Next we take the cumulative sum of the elements in array C such that the next element is the sum of the previous two elements. Now each element in a given index of array C corresponds to the number of elements less than or equal to the index. Following this we begin to populate the output array B by looping through array A from the right-most element to the first element. Then the dth element of A is taken as the index of C, and that inturn is taken as the index of B is set to the dth element of a. And then the index of array C that corresponds to the dth element of array A is decremented.

While testing the Radix Counting sort function I observed generally inconsistent running time results, unexpected trends with changing N and M values, and I was often greeted with the following error: “ hw2 1758 cygwin\_exception::open\_stackdumpfile: Dumping stack trace to hw2.exe.stackdump”. For example, the first three times I ran the algorithm with n= 5000 and M=45 I encountered the error, and then all attempts after that ran without any error. The stackdump file generated when receiving the error provides the memory address of the functions and variables involved in the cygwin exception. Compiling the “.cpp” files with the “-ggdb” flag makes these memory addresses in the stack dump file readable by the “addr2line.exe” program. The following code returns the lines responsible for the error “addr2line -f -C -e *memory address”*. Executing this points to lines 147, and 186 in sort.cpp and line 82 in main.cpp.

When the function ran without error it produced inconsistent runtimes and unexpected results. The first unexpected result is the wide range of running times of the M=45 condition when N = 2500. Figure 5 shows that the M=45 condition has a range of 50 ms to 20 ms, which is unusually high. In the same figure we see that in the second iteration the M=45 condition performs faster than the M=35 condition when the opposite is expected.

*Figure 5: Radix Counting sort N = 2500 Varying M Values*

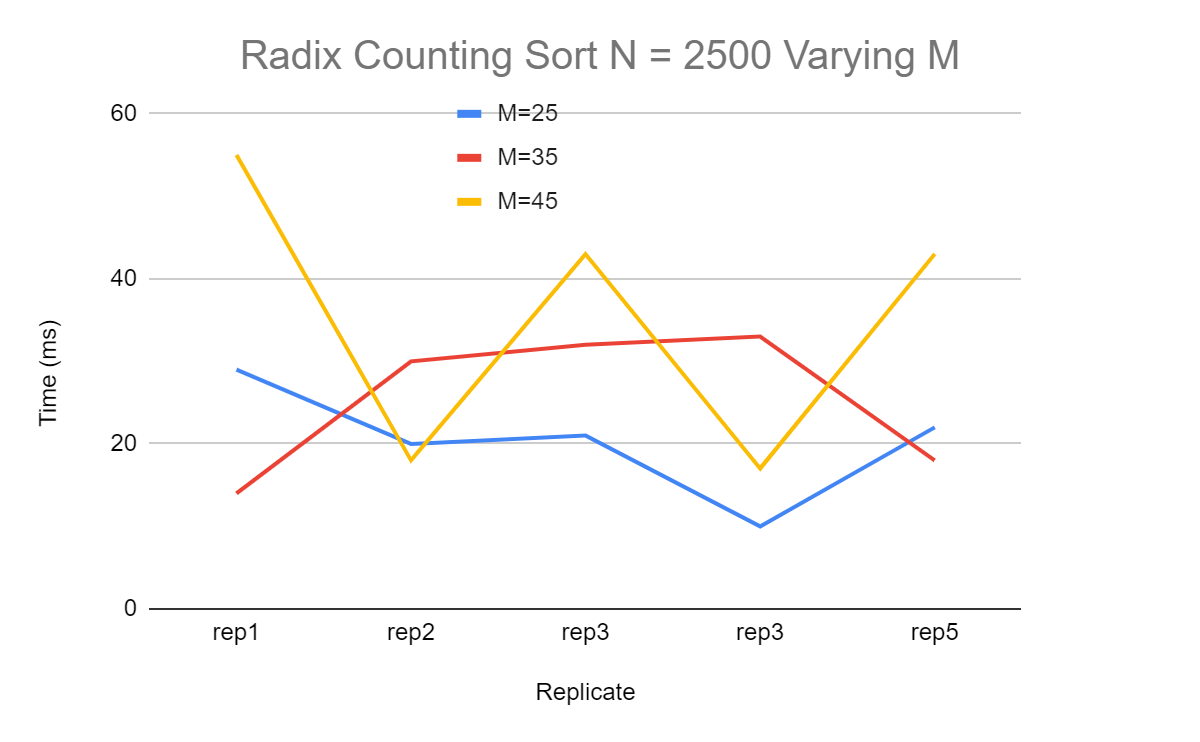
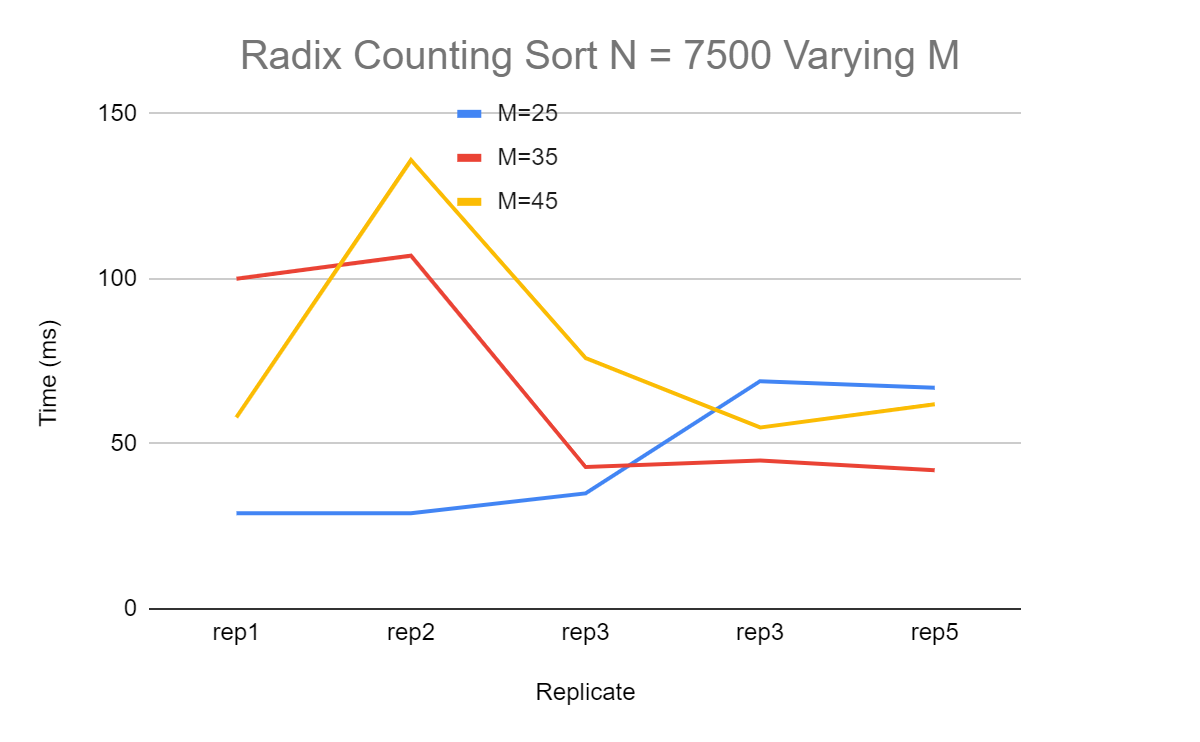


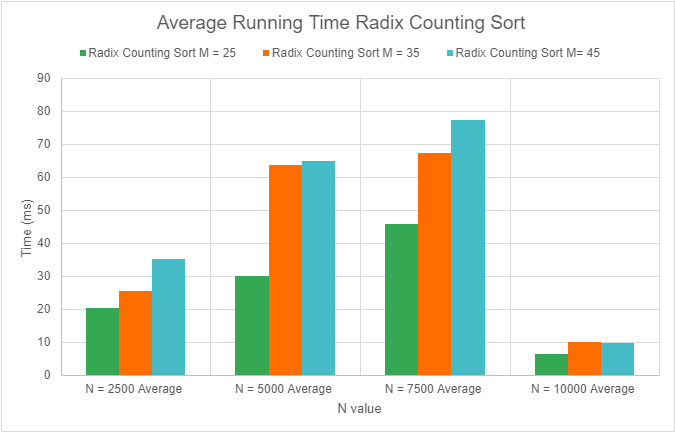
Figure 6 depicts more unexpected results from the algorithm. Here we see that the larger N value of 7500 increases the running time as expected. However on the fourth and fifth attempt, all conditions nearly converge and have runtimes such that the M=25 condition is the slowest, and M= 35 is the fastest.

*Figure 6: Radix Counting Sort N=7500 Varying M Values*

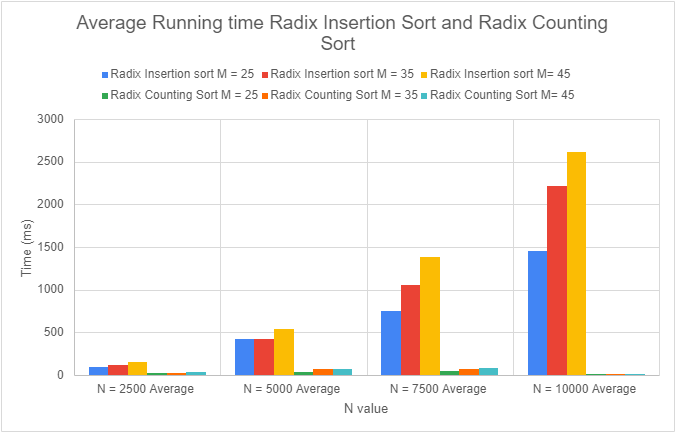


The most notable unusual behavior occurs when we run the algorithm with N=10000 over all the M conditions. Figure 7 compares the average runtime of each of the conditions and shows that the algorithm seems to run the fastest with the largest input as each of the M conditions has an average of 10 ms or less. Again this is unusual and is the opposite of what is expected. From this unusual behavior we can clearly see that the Radix counting sort algorithm is not running at the expected running time of 𝚯 (n+k).

*Figure 7: Average Running Time of Radix Counting Sort All Conditions*



*Figure 8: Average Running Time of All Conditions Radix Insertion Sort and Radix Counting Sort*



The average runtime of all 5 iterations for each of the conditions from both algorithms is shown in Figure 8. According to the figure, Radix Counting sort out-performs Radix Insertion sort over all input sizes, and this agrees with the comparison of the runtimes of the algorithms.

Overall, the Radix Insertion sort implementation agrees with the running time of 𝚯 (n2) as demonstrated by the running times and the expected trends. The implemented Radix Counting sort, however, does not agree with the 𝚯 (n+k) runtime, fails to correctly sort the input, and produces inconsistent runtimes and unexpected trends.