1. Solve the Following Recurrence

Recurrences: Solve the following recurrences using the substitution method. Subtract off a lower-order term to make the substitution proof work or adjust the guess in case the initial substitution fails.

1. $T(n) = T(n-3) + 3\lg n$. Our guess: $T(n) = O(n \lg n)$.
   Show $T(n) \le cn \lg n$ for some constant $c > 0$.
   (Note: $\lg n$ is monotonically increasing for $n > 0$)
2. $T(n) = 4T\left(\frac{n}{3}\right) + n$. Our guess: $T(n) = O(n^{\log_3 4})$.
   Show $T(n) \le cn^{\log_3 4}$ for some constant $c > 0$.
3. $T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + T\left(\frac{n}{8}\right) + n$. Our guess: $T(n) = O(n)$.
   Show $T(n) \le cn$ for some constant $c > 0$.
4. $T(n) = 4T(\frac{n}{2}) + n^2$. Our guess: $T(n) = O(n^2)$.
   Show $T(n) \le cn^2$ for some constant $c > 0$.

3.) $T(n) = (\frac{n}{2}) + T(\frac{n}{4}) + T(\frac{n}{8}) + n$

Guess: $T(n) = O(n)$

$T(n) \le cn$ for constant $c > 0$

$$
\begin{aligned}
T(n) &= (\tfrac{n}{2}) + T(\tfrac{n}{4}) + T(\tfrac{n}{8}) + n \\
&\le c(\tfrac{n}{2}) + c(\tfrac{n}{4}) + c(\tfrac{n}{8}) + n \\
&\le n(\tfrac{c}{2} + \tfrac{c}{4} + \tfrac{c}{8} + 1) \\
&\le c'n
\end{aligned}
$$

Initial guess is correct.

4.) $T(n) = 4T(\frac{n}{2}) + n^2$

Guess: $T(n) = O(n^2)$

$T(n) \le cn^2 \quad c > 0$

$$
\begin{aligned}
T(n) &= 4T(\tfrac{n}{2}) + n^2 \\
&\le 4c(\tfrac{n}{2})^2 + n^2 \\
&\le \cancel{4}c * \frac{n^2}{\cancel{4}} + n^2 \le c * n^2 + n^2 \\
&\le n^2(c+1)
\end{aligned}
$$

$c' = c+1$

$T(n) = c'n^2$

Therefore $T(n) = O(n^2)$ is proven because $c'n^2$ is $c'>$

Scanned with CamScanner

Part 1

Recurrences

1.) $T(n) = T(n-3) + 3\lg n$

Guess: $T(n) = O(n\lg n)$

$T(n) \leq cn\lg n$ constant $c > 0$

$$
\begin{aligned}
T(n) &= T(n-3) + 3\log n \\
&\leq c\cdot(n-3)\log(n-3) + 3\log n \\
&\leq c(n-3)\log n + 3\log n \\
&\leq c(n-3)\log n + 3\log n \\
&\leq cn\log n - 3\log n + 3\log n
\end{aligned}
$$

$T(n) \leq cn\log n$

Based on substition, I found that $T(n) \leq cn\log n$ forconstant proofed the guess $T(n) = O(n\log n)$.

2.

$T(n) = 4T(\frac{n}{3})$

Guess: $T(n) = O(n^{\log_3 4})$

$T(n) \leq cn^{\log_3 4}$ for some constant $c > 0$

$$
\begin{aligned}
T(n) &= 4T(\tfrac{n}{3}) + n \\
& 4c(\tfrac{n}{3})^{\log_3 4} + n \\
& \frac{4c}{3^{\log^4_3}} n^{\log^4_3} + n
\end{aligned}
$$

$T(n) \leq \frac{\cancel{4}}{\cancel{4}} c \cdot n^{\log^4_3} + n$

$\leq c \cdot n^{\log^4_3} + n$ ~~holds~~ true $\neq$ $\boxed{n^{\log^4_3} > n}$

$T(n) \leq c\, n^{\log^4_3}$

Based on the proof $c \cdot n^{\log^4_3}$ for constant, which means the guess of $O(n^{\log_3 4})$ is true.

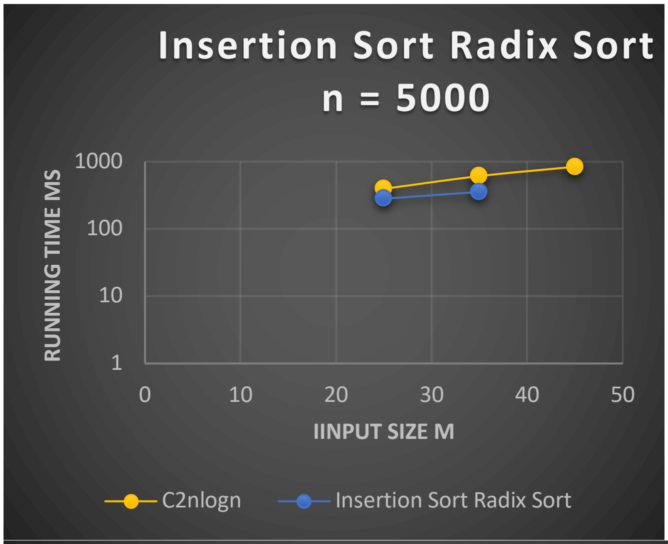
Part 2: Radix Sort on strings

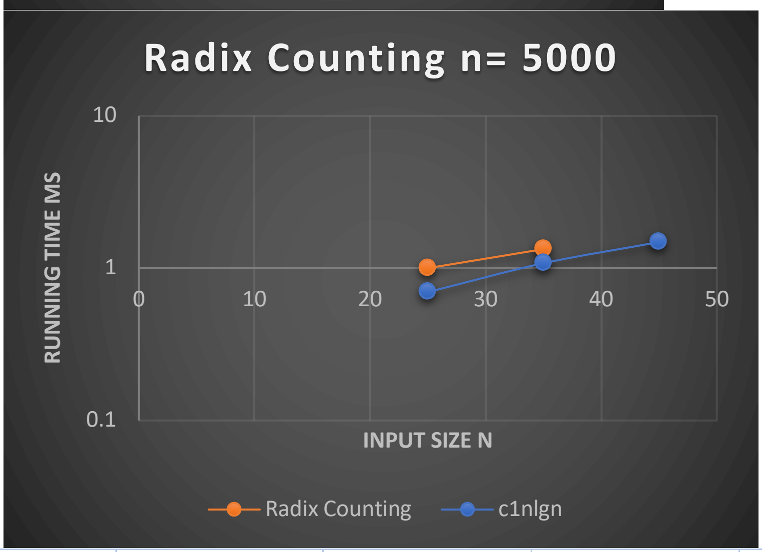
Abstract

The purpose of this project was to implement both an insertion Sort Radix sort algorithm as well as a Counting Sort Radix Sort Algorithm and to compare both of these algorithms running time complexity. For this project, there was a native Insertion Sort that was given , and I implemented the Insertion Sort_digit which sorts based on a given array of strings in accordance with the character at position d. In this specific algorithm the length of string was used array Alen to determine if the digit d existed. This was also utilized with the Counting Sort Radix Sort.

Results

Radix Sort n=5000



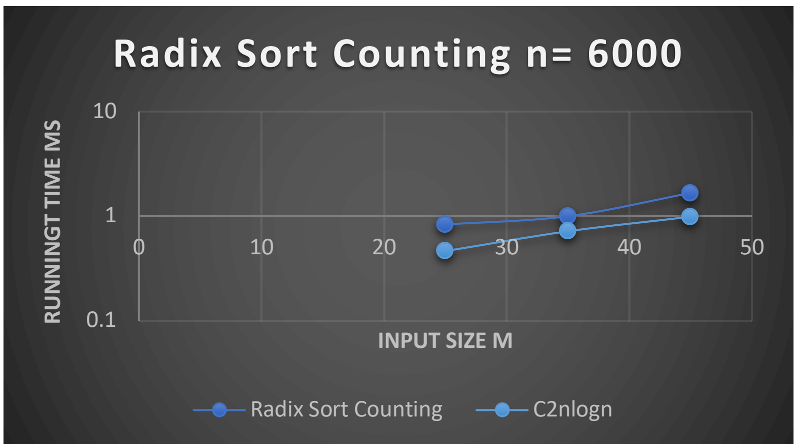


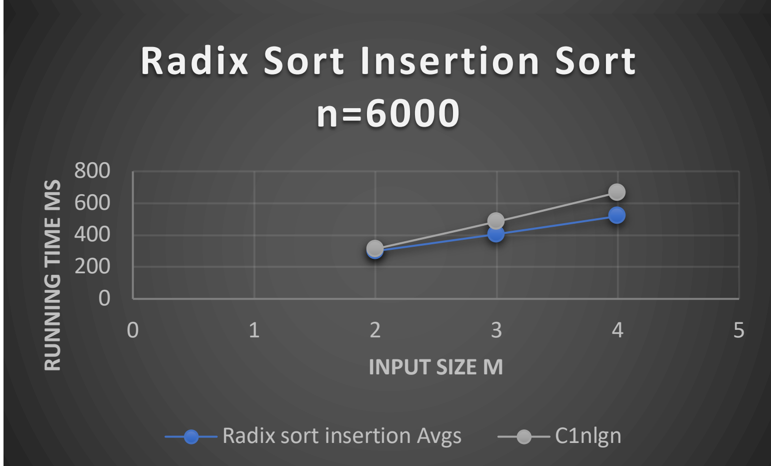
N=5000	Insertion Sort Radix Avgs	Counting Sort Radix	C1nlogn	C2nlogn
M=25	**207.2**	**.833**	0.696578428	4643.85619
M=35	**281.4**	**1**	1.077149434	7180.996224
M=45	**355.8**	**1.333**	1.482800336	9885.335573

At the values Radix sort with an n value of 5000 something weird was happening with displaying the graphs, the Insertion Sort Radix Avg where so large in the running time compared to the Counting Sort that plotting the C2nlogn and Insertion sort did not give me very good data. Due to that I parsed out the data.

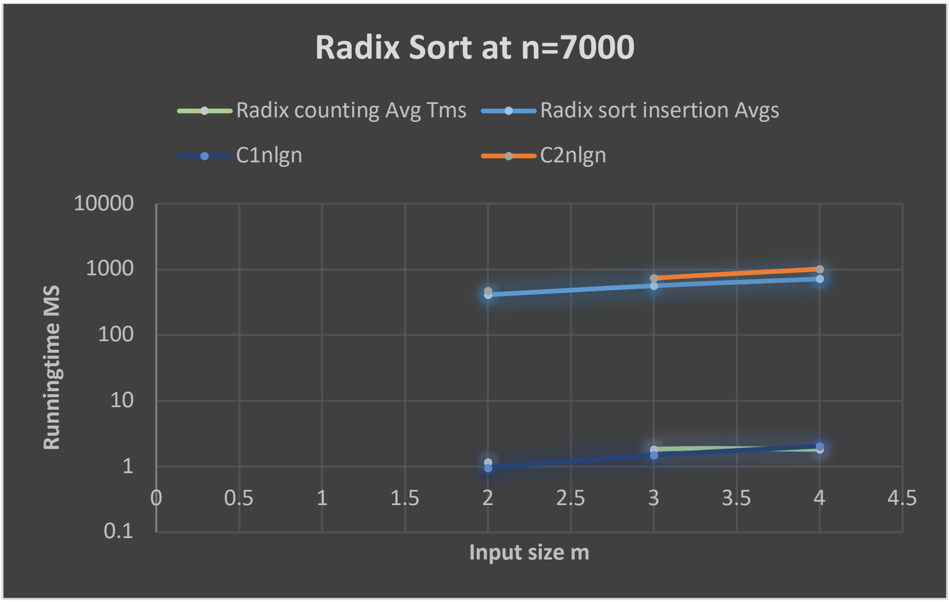
N=6000	Insertion Sort Radix Avgs	Counting Sort Radix	C1nlogn	C2nlogn
M=25	300	0.833	313.460293	0.46438562
M=35	406	**1**	484.717245	0.71809962
M=45	519	1.6667	667.260151	0.98853356

At n=6000 I ran into the same problem with the analysis, I parsed out the graphs for Insertion Sort Radix and Counting as the running time was so much larger, in comparison to each other that it would through off the graph.

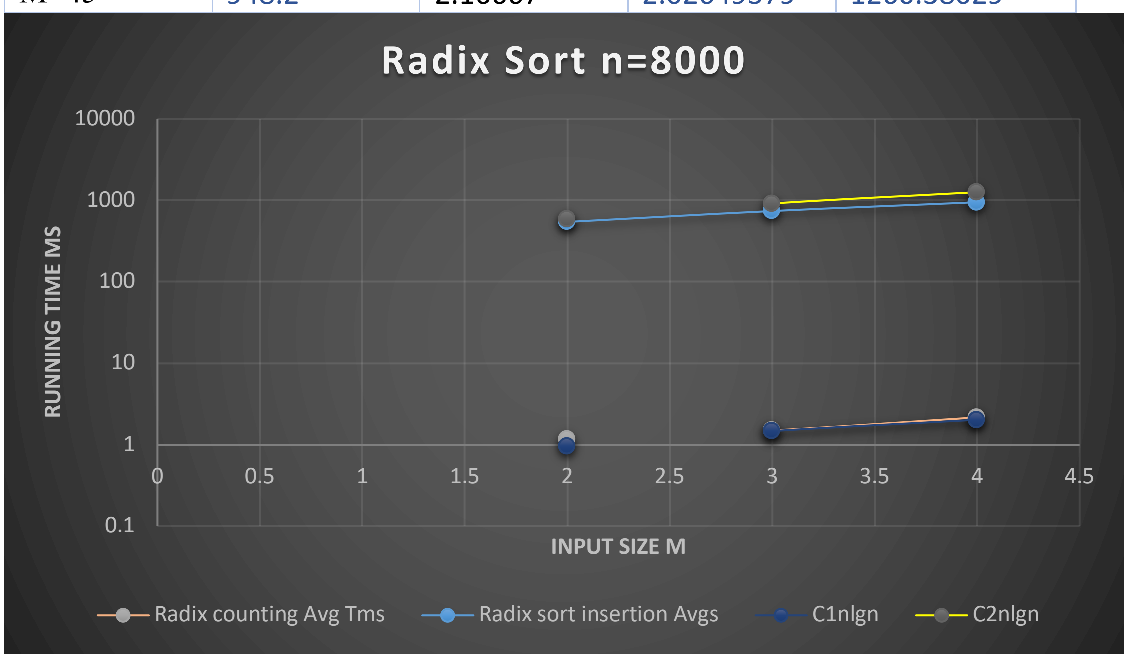




N=7000	Insertion Sort Radix Avgs	Counting Sort Radix	C1nlogn	C2nlogn
M=25	411.4	1.16667	0.96360016	475.995259
M=35	564.6	1.8333	1.49005672	736.052113
M=45	717.6	1.8333	2.05120713	1013.2469



N=8000	Insertion Sort Radix Avgs	Counting Sort Radix	C1nlogn	C2nlogn
M=25	543.4	1.16667	0.95199052	592.091664
M=35	739.6	1.5	1.47210423	915.577019
M=45	948.2	2.16667	2.02649379	1260.38029



Discussion

Comparing the running time between the two algorithms it became apparent that Insertion sort Radix held true to O(nk) where n is the length of array, k is the maximum digits. The insertion sort running time definitely is a more worst case scenario. When implementing this algorithm in the beginning I had some issues with it performing like it does, and it became apparent that at its worst case the running time for the radix with insertion sort is d(n^2). The data shows for the insertion sort radix that it does go up in value fairly linear as the input size is increased. The performance of the insertion sort radix was so significantly different then the counting radix, that in the lower n values, the graphs had to be parsed out to make it represent correctly with pairing it with the c1nlogn and c2nlogn.

The Counting Sort was much more efficient than the Insertion sort, I would say that in this case comparing both of Algorithms the Counting sort does perform at a best case time complexity of Ω(nk). For the most part there are not a lot of resources used when running this algorithm on my computer. The data shows that with the n=8000 at m=45 the running time still stays pretty low at 2.167.

Conclusion

In conclusion, I can come to the analysis that Counting Sort was much more efficient then Insertion sort with Radix Sort. Both of the algorithms hold true to its best and worst case scenario with Counting sort alone following the O(n) and usually Insertion sort behaves on the best case as O(n) as well but tying it with radix sort clearly showed that the Counting Sort Radix is much more sophisticated as the insertion radix. The Insertion Radix combination in this case does perform at its worst case of O(n^2).