Sorting Algorithm Analysis:
Radix Sort

Advanced Topics in Computer Science

Alex Xie

2022-03-21

0. Abstract

The radix sort algorithm programmed for this assignment had a theoretical “Big o” of O(ℓ*(n+k)), ℓ being how many digits are in each number, n being the array size, and k being the number of values each digit can have.[1] Radix sort should be able to outperform O(n^2) such as insertion sort and selection sort, and this was confirmed after testing with arrays of significant sizes ranging from 100,000 to 1,000,000 items. Radix sort also significantly outperformed a standard quicksort on these arrays as radix sort’s time complexity appeared to be a linear time while quicksort was polynomial time. However, Python’s built-in sort() method significantly outperformed radix sort as Python uses timsort, a combination of insertion sort and merge sort. Also included is a visualization of radix sort using Processing. The visual representation of the sorting algorithm depicts the stages of the sort as it goes through the different stages. All of the code for this project can be found on Github at: https://github.com/axie22/radixSort

1. Introduction

Sorting algorithms are present in everyone’s daily life. From browsing clothes online to watching TV, sorting algorithms have been implemented and optimized for different tasks. However, not all sorting algorithms are created equal as some are more efficient and take up less memory. To measure an algorithm’s time complexity, computer scientists use Big O notation to see how the algorithm will scale as there are more items to sort. This paper compares three different sorting algorithms with a focus on radix sort.

First, radix sort’s strategy will be laid out using both diagrams and explanations followed by the source code of radix sort’s implementation in Python. Next, radix sort’s performance will be analyzed, first using Big O notation to determine the theoretical predicted efficiency then experimentally using Python’s `time` module and arrays of increasing sizes. The efficiency of this sorting algorithm will then be graphed and compared to four other sorting algorithms. The paper will then move to a quick extension of a visualization of radix sort using processing. The source code for this program will also be provided as it is slightly different. A discussion of the results and work will then take place followed by a summary concluding the paper.

2. The Sorting Algorithm

Though radix sort is not commonly used, the sorting algorithm does have its special use cases; it typically shines when data can be sorted lexicographically, meaning in dictionary order. Radix

[1]https://www.interviewcake.com/concept/java/radix-sort#:~:text=Radix%20sort%20takes%20O%20(%20%E2%84%93,values%20each%20digit%20can%20have.

sort works by going through the different number places starting with the least significant digit and sorting the array in ascending order. The sorting algorithm continues this, going through the different number places until the largest number has been sorted. In the implementation, a secondary array called "buckets" is used. Because we are using base 10 numbers, the "buckets" array has a length of 10, each spot coinciding with the values from 0-9. As the program traverses through the array, the values are put into the "buckets" array based on the number that is being examined. For example, if the "ones" number place was being examined for the number 72, it would go in buckets[2]. After the entire list has been traversed and the "buckets" array is full, the values in the "buckets" array will replace those in the original array and the process will continue until the entire list has been sorted. The following diagram illustrates how radix sort would function on an unsorted array.

Original unsorted array: `nums = [9, 31, 96, 100]`

Identify the largest number: `[9, 31, 96, 100]`

While the largest number has not yet been sorted, go through the different number places and sort the array starting from the ones place:

Create a secondary 2d array with a size of 10 x 1:
`buckets = [[], [], [], [], [], [], [], [], [], []]`

Go through the unsorted list examining the digit in the ones place and add the numbers to the buckets that coincide with the value being examined:

```
                    [9, 31, 96, 100]
                            ↓
 buckets = [[100], [31], [], [], [], [], [96], [], [], [9]]
              0      1    2    3  4    5    6     7   8   9
```

Add the arrays from the bucket back into the original array and repeat the process with the tens place:

`nums = [100, 31, 96, 9]`

Create a secondary 2d array with a size of 10 x 1:
`buckets = [[], [], [], [], [], [], [], [], [], []]`

Go through the unsorted list and examine the digit in the tens place and add it to the bucket that coincides with its value:

```
                    [100, 31, 96, 9]
                           ↓
 buckets = [[100, 9], [], [], [31], [], [], [], [], [], [96]]
               0       1   2    3     4   5   6   7  8     9
```

(Note: if a digit does not have a tens place, then it is assigned a 0 value.

Add the arrays from the bucket back into the original array and repeat the process with the hundreds place:

```
nums = [100, 0, 31, 96]
```

Create a secondary 2d array with a size of 10 x 1:

```
buckets = [[], [], [], [], [], [], [], [], [], []]
```

Go through the unsorted list and examine the digit in the tens place and add it to the bucket that coincides with its value:

```
                    [100, 0, 31, 96]
                           ↓
 buckets = [[0, 31, 96], [100], [], [], [], [], [], [], [], []]
             0             1      2   3   4   5   6   7  8    9
```

(Note: if a digit does not have a tens place, then it is assigned a 0 value.

Add the arrays from the bucket back into the original array and repeat the process with the hundreds place:

```
nums = [0, 31, 96, 100]
```

The number of stages that radix sort must go through is dependent on how large the largest value is. In this case since the largest value was 100, the sort had to go through three different stages to account for the ones, tens, and hundreds place.

3. Source Code

The following is the Python implementation of radix sort used for the data collection and analysis for this project:

```
def radixSort(nums):
    maxDigit = max(nums) #finds the largest number in the array
    digit = 1 #start with the ones place
    while maxDigit >= digit:
        #10 buckets for the digits 0-9
        buckets = [[] for i in range(10)]
        for num in nums:
            # place the number in the corresponding bucket based on
the value of the digit in the number place being examined
            buckets[(num // digit) % 10].append(num)
    nums = []
    for bucket in buckets:
        #add the numbs in the bucket back into the empty list
        nums.extend(bucket)
    digit *= 10 #move to the next numbers place
return nums
```

4. Big-O Analysis

Now that radix sort's general strategy and source code have been shown, we can estimate radix sort's runtime using Big-O analysis. The average case, worst case, and best case runtimes will be examined in the following section.

Average Case: The average case complexity occurs when the numbers to be sorted are somewhat evenly distributed. This means that with each pass through the different number places, the algorithm will steadily sort the numbers at a similar rate. Because radix sort is dependent on, "d", how many number places the largest number has and "b", the base system we are using (in this case base-10), radix sort has an average time complexity of O(d(n+b)). This makes it a linear sorting algorithm for its average case complexity.

Worst Case: The worst case complexity occurs when all of the numbers in the array have the same number of digits, except one outlier with a significantly larger number of digits. This is because radix sort sorts the numbers in stages going through each number place before moving onto the next. If all the elements were to be already sorted except for one digits, the program would have to reiterate the mostly sorted array many times before it is completely sorted. This inefficiency causes the algorithm to degrade to an O(n^2) time complexity.

Best Case: The best case occurs when all the values in the unsorted array have the same number of digits. This is because the array will not have to continually shuffle through already sorted numbers. This would skim down the runtime of the algorithm to O(dn) where “d” is how many number places the largest number has and “n” is the size of the array.

5. Performance Data

To check whether or not the sorting algorithm worked correctly, we can use Python’s `time` module. The data below shows the times it took for radix sort to sort arrays of increasing size starting from 100,000 and ending with 1,000,000 using a step of 100,000 each trial. The unsorted lists were randomly generated using Python’s `random` module with values varying from 0 to 100,000.

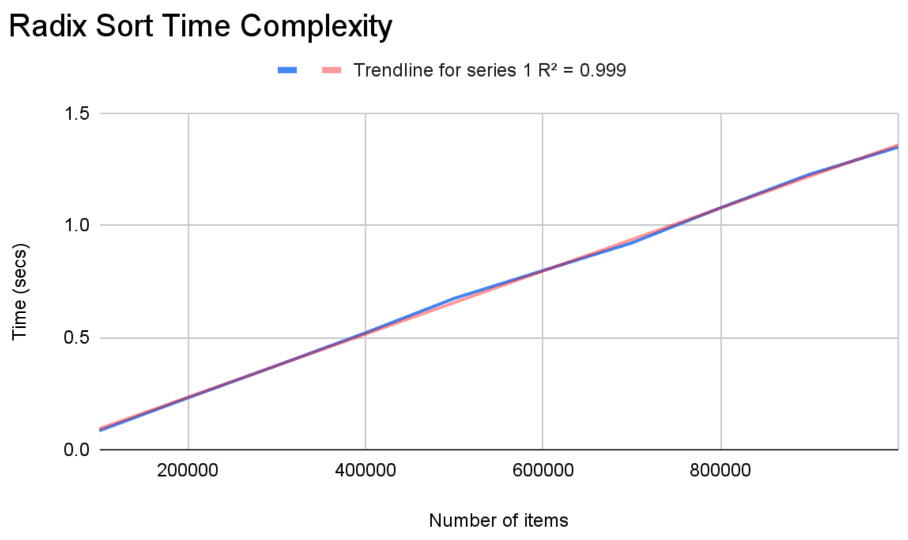
Number of Items	Time (secs)
100,000	0.08432412148
200,000	0.2309477329
300,000	0.3747389317
400,000	0.5207901001
500,000	0.6741430759
600,000	0.7982378006
700,000	0.9213268757
800,000	1.078088045
900,000	1.226894855
1,000,000	1.348120213

The data table below shows the times it took radix sort to sort the same size arrays as those above; however, instead of numbers ranging from 0 to 100,000, we have decreased the variance to 0 to 1,000. As expected, the time it took to sort the array decreased.

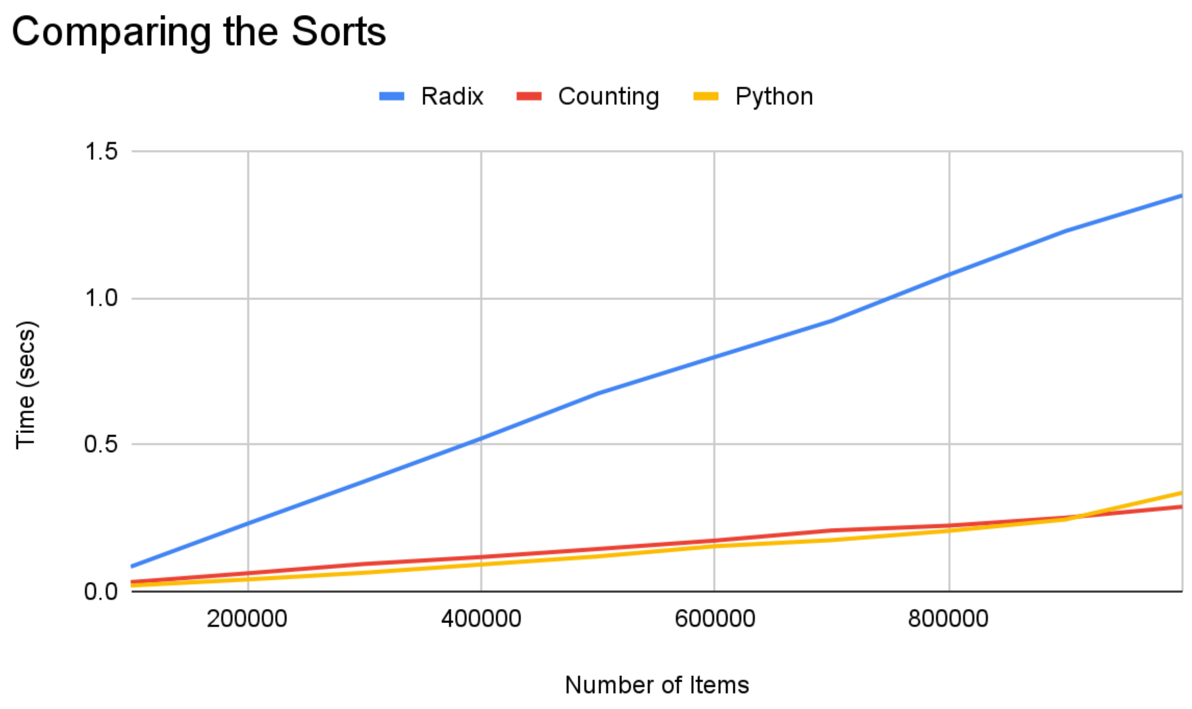
Number of Items	Time (secs)
100,000	0.06601119041
200,000	0.1483228207
300,000	0.2271270752
400,000	0.3134150505
500,000	0.3921306133
600,000	0.4706380367
700,000	0.5640277863
800,000	0.64635396
900,000	0.7417867184
1,000,000	0.8314759731

6. Graphical Results

After graphing the data using Google Sheets, it is clear that radix sort is not exponential nor quadratic, meaning that it is viable sort as it can scale fairly well. The predicted graph for radix sort was linear as derived in section for of the Big-O analysis where we derived that radix sort has an average runtime of O(d(n+b)). After using Google Sheet's "Trendline" feature, we were able to see that a linear graph fit the data fairly well with an R^2 value of 0.999.

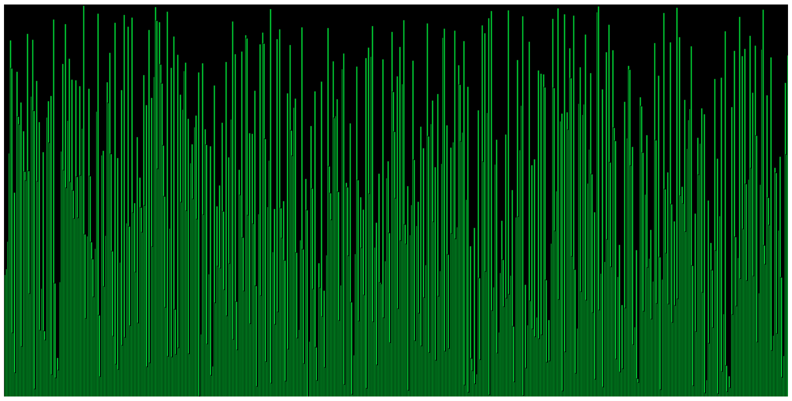


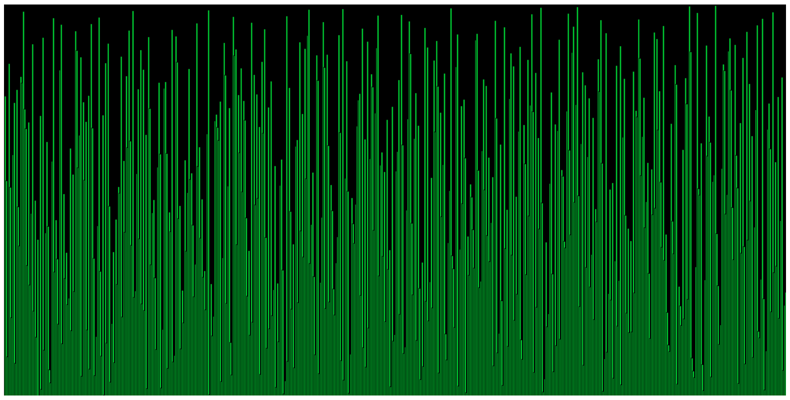
The following data table compares three different sorting algorithms: Radix Sort, Counting Sort, and Python's built-in sorting algorithm (timsort). As expected, Python's built-in sort() performed the best; however, counting sort, a sort very similar to radix sort, performed very well as well.

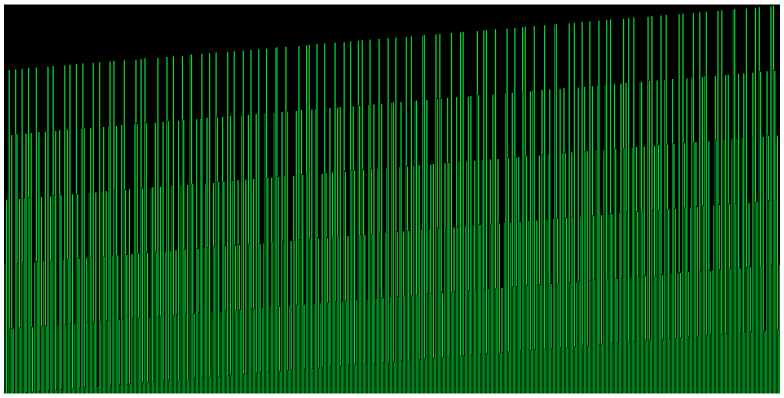


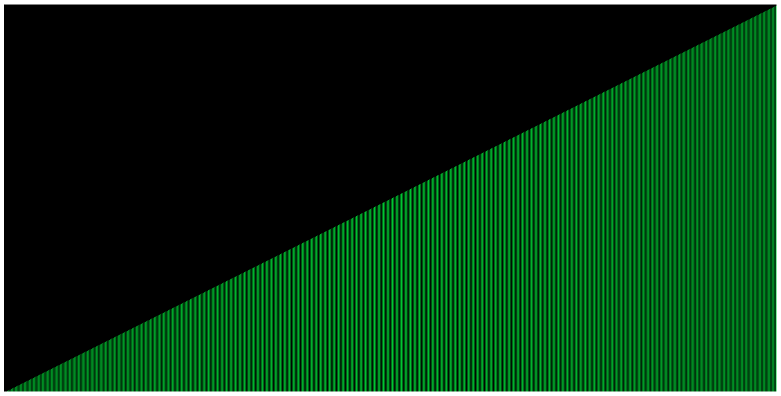
7. Extension: Visualization

The goal of this extension was to provide a visual representation of the process of radix sort. To achieve this, we used Processing, a free graphical IDE that can be used in Python. The height of the rectangles in the program are representative of the size of the value. The code for this program is similar to the standard code seen in section 3; however, certain modifications were made in order for the visual aid to work. The array had a length of 600 with values ranging from 0-600. Because of this, there are three clear stages to the sort as the algorithm goes through the ones, tens, and hundreds place. The code can be found in the github repository linked below: https://github.com/axie22/radixSort









8. Discussion

The results in this paper show that radix sort is a fairly effective algorithm and can be used to sort large data sets; however, it is still not nearly as efficient as Python's `sort()` or even counting sort, a sorting algorithm that is thought to be a "simpler" version of radix sort. Radix sort could be optimized to fit specific situations and even coded in a faster language such as C++ which allows for more control and optimization . However, the same can be said for other sorting algorithms that are inherently faster such as timsort; therefore, radix sort is sparsely used in the real world.

9. Summary

This paper first introduced radix sort as a fairly efficient sorting algorithm that is not commonly used in the real world but is still viable. A diagram and explanation of the strategy of this sorting algorithm was shown followed by source code with comments providing narration on what was happening in the program. Next, we examined the average, worst, and best time cases of radix sort using Big-O notation and determined the sort to be linear. We then verified this with performance data from the source code using Python's `time` module. Radix sort was able to significantly outperform simple $O(n^2)$ sorting algorithms such as selection and insertion sort; however, was beaten by Python's `sort()` as well as counting sort. To attain a more visual understanding of our data, we used Google Sheets to plot the data points where we were clearly able to see the regression line of radix sort as well as how it compared to other sorting algorithms. The quick extension of this paper covered a visualization of radix sort using Processing. Here we were able to see the different stages of the sort as it worked through a shuffled array. Finally, the results of the project were discussed and it was determined that while there are niche use cases for radix sort, more efficient sorting algorithms such as quicksort and timsort will almost always be the safer and more efficient bet.

10. References

Wikipedia article on radix sort: https://en.wikipedia.org/wiki/Radix_sort

Processing helper functions and references: https://processing.org/reference/

Radix sort's sorting strategy: https://www.geeksforgeeks.org/radix-sort/