Radix Sorting Algorithm

Bryant Passage

November 2019

1 Introduction

With the age of computing in full rise and computers increasingly becoming faster and more complex, algorithms are still at the heart of computer science. Whether from data science analysis to bioinformatic processing, computer scientists are still looking for better and more efficient algorithms to help break down some of the toughest problems in modern day computing. One of the most fundamental and constantly researched algorithm in computing is sorting.

Sorting algorithms are necessary when dealing with data sets that need to be ordered in some form, whether that be lexigraphically or numerically. Without sorted data sets, simple functions such as searching elements or data analysis becomes very difficult and inefficient. Working with a sorted list is much more convenient and more efficient. Faster and efficient algorithms allows for more time allocation into other problems that occur after data analysis over a set is finished. Therefore, it is in a programmer's best interest to find the fastest sorting algorithm over a universal set. There are, of course, caveats to such an ambitious goal.

In this report, we will discuss the radix sort algorithm – a non-comparison based sorting algorithm that sorts based on the element's radix and utilizes a subroutine that places the order of each radix.

2 History

Versions of radix sort have been used early on in the computing era. Around 1887, radix sort can be found in tabulating machines by the works of Herman Hollerith. Radix sorting algorithms really came into common use as a way to sort punched cards starting around 1923. However, even though radix sort algorithms were used particularly on physical machines to sort through programmed punch cards, computerized radix sorts were not used as often because of its perceived need of variable memory, since radix sort is not constant in memory, which will be discussed in a later section.

3 Algorithm

The term "radix" refers to the number of unique digits of numeration. Radix is also referred to as the base of a number system as well (i.e. decimal, binary, hexadecimal, etc.). From this definition, the reader can infer that this sorting algorithm will be based on the ordering of each digit of an element, most likely an integer. Radix sort utilizes another sorting algorithm to arrange each element's digit. The difference with the algorithm of radix sort is that radix sort utilizes the other sorting algorithm to sort each radix of the element and will iterate over until the maximum element's digit length is reached. Since radix sort depends

on another sort to complete the algorithm, the performance and efficiency of radix sort depends on the sub-sorting algorithm as well. For this report, counting sort is paired with radix sort to provide an efficient, stable sorting algorithm. To better understand radix sort, counting sort must be understood.

3.1 Counting Sort

Counting sort iteratively reads through each element in a given list and counts the number of occurences of each element via the index of a separate array, which will be called the counting array from now on. Then, each index of the array will be the added by the number at the index before it starting with the first two elements in the counting array. For example, a list of the integers $\{4, 5, 3, 6, 7, 9, 6, 3, 2\}$ will be counted in the counting array as $\{0, 0, 1, 2, 1, 1, 2, 1, 0, 1\}$. Then, a loop will go through all the numbers in the counting array and add each number by the number in the previous index. This brings the counting array to $\{0, 0, 1, 3, 4, 5, 7, 8, 8, 9\}$. Then, the original input array and the counting array are used to sort the input array. The first number of the input array is 4. The fourth index of the counting array is the integer 4. That means that the number 4 resides at the (4-1)th index of the sorted array. Lastly, the fourth index of the counting array is decremented. This brings the counting array's contents to be $\{0, 0, 1, 3, 3, 5, 7, 8, 8, 9\}$. Following this principle for the rest of the numbers in the elements in the input array, the sorted array would be $\{2, 3, 3, 4, 5, 6, 6, 7, 9\}$ and the counting array would be $\{0, 0, 0, 1, 3, 4, 5, 7, 8, 8\}$. This is the fundamentals of counting sort.

3.2 Radix Sort (utilizing Counting Sort)

Explained above, radix sort utilizes a separate sort to sort through each digits place of each element in the list. To complete the radix sort algorithm, radix sort applies counting sort to each elements' digits. Once the last digit passes through the counting sort, then the list is completely sorted. To give an example, take the list $\{222, 121, 221, 181\}$. Radix sort will use counting sort starting with either the least significant bit or the most significant bit. For this example, the list will be sorted starting from the least significant bit. After the first pass of the algorithm, the list is sorted as $\{121, 221, 181, 222\}$. Radix sort will then sort the second digit. This will result in the array $\{121, 221, 222, 181\}$. Lastly, after sorting the most significant bit of the list, this would result in the sorted array $\{121, 181, 221, 222\}$. This concludes the algorithm of radix sort using counting sort.

The number of passes that radix sort executes is the maximum element's number of digits. From the example above, radix sort performed three passes on the set above because the max element (222), has three digit places. Because of this observation, it becomes clear that the max element in a given list is a determining factor in the performance of this sorting algorithm. The the next section, the performance of radix sort will be analyzed and compared to other sorting algorithms.

4 Performance

4.1 Time Complexity

Because radix sort's performance relies on the performance of its sub-sorting algorithm, counting sort's performance must also be analyzed. The running time complexity of counting sort is $O(n + k)$ where k is the maximum number of the set. k can also be referred to as the number of keys required to sort the list. This makes sense as, at first, counting sort traverses through the input data list and counts the number of occurrences of the elements in the list and stores those counts in a counting array. This takes $O(n)$ time.

Then, the input data list and counting array list are used to find the correct place in the sorted list. This process takes $O(k)$ since k is the length of the counting array list. This leads to counting sort to have a time complexity of $O(n + k)$.

Now, using counting sort as the routine for radix sort, the time complexity of radix sort comes to be $O(d(n+k))$, where k is the base of the number system and d is the number of passes required to completely sort the list. The variable d can also be written out as $log_k(max)$ where max is the maximum value/element in the list to sort. This allows the reader to better visualize that the running time complexity of radix sort is truly $O((n + k) \cdot log_k(max))$.

4.2 Memory Complexity

Radix sort's worst-case space complexity turns out to be $O(n + k)$. This is due to the fact that counting sort requres a separate array of keys/buckets to store the number of occurrences of certain keys from the original list. Thus, this creates an n-number array for the original array and a k-number array for the key array thus making the space complexity of radix sort $O(n + k)$.

4.3 Radix Sort vs. other Comparison Sorts

The best that comparison sorts can do in terms of time complexity is $O(nlogn)$ due to the fact that comparison sorts need to compare each element in relation to the other elements in the set/list. On the other hand, radix sort can perform better than comparison sorts, boasting as low as an $O(n)$ time complexity to sort a given data set. If, for example the max possible value in a given unsorted list was equal to the base of the numbering system, then the algorithm could sort a list with n elements in $O(n + k)$ time. This is better than the fastest comparison sorts such as merge sort and quick sort. However, if the maximum element in the set was some number n^c, for c being a constant, then this radix sort would perform the sort in $O(nlog_k n)$ time, not any better than $O(nlogn)$ comparison sorts. Therefore, the performance of radix sort heavily depends on the range of elements present in the list to be sorted.

4.4 Stability

One of the main attraction of radix sort is its stability factor. A sorting algorithm is stable if duplicate keys in the unsorted list maintains the same order in the sorted list. Take the set $\{C, A, B, A\}$. There are two A's present in the set. We can distinguish them by using subscripts which makes the set become $\{C, A_1, B, A_2\}$. A stable sort would sort the elements so that A_1 still precedes A_2. Therefore, the sorted list would be $\{A_1, A_2, B, C\}$. Counting sort is generally implemented as a stable sort and therefore allows radix sort to also be a stable sorting algorithm unlike comparison sorts such as heap sort, quicksort, selection sort, etc. Note that merge sort and insertion sort are also stable.

5 Applications

Sorting algorithms are some of the backbones of computer science and computing research and are used every where where data sets are analyzed and formatted. Radix sort is no different in helping us organize and analyze data in a quick and efficient manner. However, the applications of radix sort extend beyond the organization of integer/character sets. Radix sort is used to construct suffix arrays in $O(nlogn)$ time while also maintaining to be stable. Suffix arrays are lists of all suffixes of a character string. For example, the suffix set for "apple" would be $\{apple, pple, ple, le, e\}$. However, the set also needs to be alphabetized and

instead represented with integers that represent the index of the first letter of the suffix from the original character string. Therefore, the suffix array for "apple" would be $\{0, 4, 3, 2, 1\}$. The full implementation of suffix array construction is outside of the scope of this paper, however, radix sort plays an important role in suffix array contruction by ranking the suffixes by their ASCII value, which will lead to the ordering of the suffixes, all while maintaining stability of the subset of suffixes.

Radix sort can also be seen in parallel computing as a means of a parallel sorting algorithm. Here, each processor/thread is given a parition of a list to sort and are told to locally sort the paritions. Then, each processor is commanded to retrieve certain number of buckets/keys. This is the redistribution stage. Then, the process is repeated with the same number of partitions on different processors, except this time, they locally sort the next digit of the elements. At the end, a sorted array using parallized radix sorting can be achieved with the theoretical lower running time than when running sequentially. Although this method is not completely perfect, methods are still being researched in order to make parallized radix sorting possible.

References

[1] Gupta, Shubham. "Radix Sort." *HackerEarth*, https://www.hackerearth.com/practice/algorithms/sorting/radix-sort/tutorial. Accessed 9 November 2019.

[2] He, Rolland. "PARADIS: A parallel in-place radix sort algorithm." https://stanford.edu/ rezab/classes/cme323/S16/projects_reports/he.pdf. Accessed 9 November 2019.

[3] Joshi, Vaidehi. "Getting To The Root Of Sorting With Radix Sort." *Medium*, https://medium.com/basecs/getting-to-the-root-of-sorting-with-radix-sort-f8e9240d4224. Accessed 9 November 2019.

[4] Knuth, Donald. *The Art of Computer Programming*. Vol. 3, Third Edition. Addison-Wesley, 1997. Section 5.2.5: Sorting by Distribution, pp. 168-179.

[5] "Radix Sort." *GeekforGeeks*, https://www.geeksforgeeks.org/radix-sort/. Accessed 9 November 2019.

[6] Vladu, Adrian, and Cosmin Negruseri. "Suffix arrays - a programming contest approach." *GInfo*, Nov. 2005, https://web.stanford.edu/class/cs97si/suffix-array.pdf. Accessed 9 November 2019.