Comparison Between Quicksort and Radixsort

Shariq Butt - 2/28/20

Quicksort

Quicksort is a comparison-based divide and conquer algorithm. This sort works by choosing a partition at any point in the array, which, in this experiment, will be the median after randomly shuffling the array. The elements of the array are then partitioned into subarrays based on whether they are greater than or less than the pivot. The subarrays are then sorted recursively with the same pivot process until all of the array is sorted. Quicksort requires very little additional memory as it can be done in place, which makes it efficient for multiple types of data. Quicksort has an average runtime complexity of O(nlogn) and a worst-case runtime complexity of $O(n^2)$.

Radixsort

Radixsort is a non-comparative sorting algorithm, which works by sorting elements of the array into bins based off of their radix. Elements are sorted iteratively until all of their digits have undergone sorting. Once all of the digits have been passed through the counting sort subroutine, the array is finally sorted. Radixsort requires extra memory and makes as many bins as there are digits in the largest number of the array, which means that this algorithm can not be performed in place. This algorithm can be used on any data which can be sorted lexicographically. The runtime complexity of this algorithm is O(d*(n+b)), where b is the base used for representing numbers and d is the number of digits in the input integers. This algorithm has a linear runtime complexity of O(n) when b=n and $k <= n^c$, allowing for low runtimes for wide ranges of data.

Data

In this experiment, we will use 4 datasets to demonstrate when either quicksort or radixsort has advantage. Datasets 1 and 2 will be artificially created while Datasets 3 and 4 will be found from a real source. Quicksort will be expected to run slower than radixsort for all values except for those which are very high and have a lot of digits in the maximum of the dataset.

Dataset 1 (AP Score Data, bin optimized)

Radixsort outdid Quicksort by a large margin with this dataset, and with a bin size of 15, it almost cut its runtime down to third of Quicksort's

Bin size	Quicksort runtime (μs)	Radixsort runtime (μs)
2	1319	1263
3	1319	1225
4	1319	1185
5	1319	958
6	1319	560
7	1319	537
8	1319	541
9	1319	733
10	1319	634
11	1319	537
12	1319	647
13	1319	548
14	1319	535
15	1319	529

Dataset 2 (Age distribution in Chandler)

Radixsort proved to be much quicker, generally outperforming quicksort for bin sizes greater than 8, and showed a logarithmic decrease in runtime as bin size grew larger and larger.

Dataset 3 (Order ID for Shipments)

Quicksort seemed to run faster than radixsort for this dataset and this is likely due to the fact that the number of iterations that radixsort does goes up with the number of digits in the largest value. Quicksort outperformed radixsort for almost every relevant bin size, and radixsort was only able to compete when bin size got very high, which is not feasible for machines with low amounts of memory.

Dataset 3 (Total Revenue for Shipments)

Quicksort seemed to run faster than radixsort for this dataset as well and this is likely due to the fact that the number of iterations that radixsort does goes up with the number of digits in the largest value. Quicksort outperformed radixsort for almost every relevant bin size, and radixsort was only able to compete when bin size got very high, which is not feasible for machines with low amounts of memory.

Conclusion

Radixsort proved to be much quicker than quicksort for almost all integer datasets; However, the space complexity of radixsort is most definitely a problem because with large bin sizes the algorithm became too memory intensive. While this may not be a problem for larger devices, it certainly is a problem for lower end machines and smaller hardware. Quicksort managed to do much better with datasets that we too large for radixsort to efficiently perform counting sort on. From the data collected, we can conclude that radixsort works best on small integer datasets that don't have too many significant digits, and that quicksort has a decent runtime for all integer data except for when there are a very large

number of duplicate keys. This data also didn't take into account the versatility of quicksort because quicksort can accept so many more types of data compared to radix sort.