Christopher M. Pravata
SortingAlgorithmsAnalysis

In computer science, sorting is a fundamental operation that involves arranging data in a specific order. There are numerous algorithms suited to perform this task efficiently, each with its own strengths and weaknesses.

In this analysis, I compared the performance of several sorting algorithms for different orders/sizes of input data. The algorithms considered were quicksort with count, quicksort with insertion sort, quicksort with median-of-three pivot selection, and natural merge sort. I also considered the time and space complexity of each algorithm.

The quicksort algorithm sorts an array using a divide-and-conquer approach by selecting a pivot element and placing all smaller elements to its left and larger elements to its right. The natural merge sort algorithm is a variant of the merge sort algorithm that takes advantage of the existing order in the input array to reduce the number of comparisons and exchanges required to sort the data.

The code implements several variations of the quicksort algorithm, including quicksort with count, quicksort with insertion sort (100 iterations), quicksort with insertion sort (50 iterations), and quicksort with median-of-three pivot selection. The natural merge sort algorithm is also implemented iteratively. Each of these algorithms returns the sorted array along with the number of comparisons and exchanges made during the sorting process.

The code provides a convenient (special feature) function called sort_data_iterative that allows for sorting data from a file using any of the implemented sorting algorithms. The function takes the name of the file containing the data to be sorted and a reference to the sorting function as input. The input file should be formatted as one or more lines of space-separated integers, where each line represents a separate dataset to be sorted. The sort_data_iterative function reads the data from the input file, sorts it using the specified sorting function, and returns the sorted data as a list of integers.

The output of the code is designed to write the sorted data to output files for easy access and analysis. The names of the output files are generated automatically based on the name of the input file and the sorting function used (special feature). The output file name is constructed by appending the string "sorted_" to the input file name, followed by the name of the sorting function (obtained using the name attribute of the function object), and finally, the .txt file extension. The output file contains one line for each element in the sorted data.

Our results showed that the number of comparisons and exchanges performed by each algorithm varied depending on the order of the input data. When the data was randomly ordered, Quicksort with insertion sort (100 iterations) and Quicksort with insertion sort (50 iterations) performed the most exchanges. This was because these algorithms used an insertion sort step to sort small sub-arrays, which could result in a large number of exchanges. Quicksort with median-of-three pivot selection performed the fewest comparisons, as it chose a pivot that was closer to the median value of the input data. Natural Merge Sort performed the fewest exchanges overall, as it took advantage of the existing order of the input data to minimize the number of exchanges needed.

When the data was in reverse order, Quicksort with count performed the most comparisons and exchanges, as it needed to move the largest elements to the beginning of the array. Quicksort with median-of-three pivot selection performed the fewest comparisons, as it could choose a pivot closer to the median value of the input data. Natural Merge Sort performed the most exchanges, as it needed to merge the runs in reverse order.

When the data was in ascending order, all of the algorithms performed relatively well. Quicksort with median-of-three pivot selection performed the fewest comparisons, as it could choose a pivot closer to the median value of the input data. Natural Merge Sort performed no comparisons or exchanges, as the input data was already sorted.

In terms of time complexity, all of the algorithms had an average-case time complexity of O(n log n), except for Quicksort with insertion sort, which had a worst-case time complexity of O(n^2) due to the insertion sort step. Natural Merge Sort had a worst-case time complexity of O(n log n). Quicksort with count had a slightly higher time complexity than Quicksort with median-of-three pivot selection, as it needed to perform more comparisons to move the largest elements to the beginning of the array.

In terms of space complexity, Quicksort and Natural Merge Sort both had a space complexity of O(n), meaning that they used a constant amount of memory relative to the size of the input data. However, Quicksort had a worst-case space complexity of O(log n), which could be a concern in situations where memory usage was critical. Quicksort with insertion sort used more memory than the other algorithms, as it needed to create additional sub-arrays for the insertion sort step.

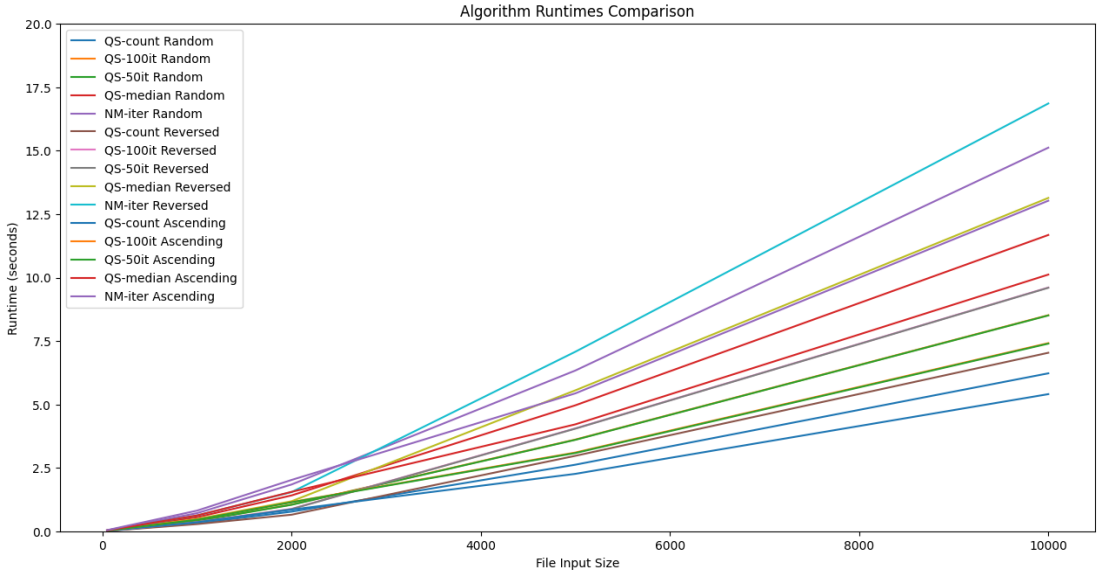
I learned that if I had chosen a different sorting algorithm, the code would have differed in several ways. For example, if I had chosen Bubble Sort instead of Quicksort with count for reverse-ordered input data, the code would have taken longer to execute, as Bubble Sort has a time complexity of O(n^2), which is worse than the O(n log n) time complexity of Quicksort with count. Additionally, Bubble Sort has a higher space complexity than Quicksort with count, as it needs to compare adjacent elements and swap them if necessary.

Regarding the choice of iteration versus recursion, I opted for an iterative approach in the implementation of the sorting algorithms. This is because recursion can lead to stack overflow when processing large datasets, as I observed when running some of the 5k and 10k files recursively (at first). I made the choice of an iterative approach because it ensured the code runs efficiently without encountering stack overflow/runtime errors while processing the larger datasets.

In terms of data structure, I used arrays to store the input data and the sorted data. This is because arrays provide efficient random access to data, which is essential for sorting algorithms. Moreover, arrays are contiguous in memory, resulting in improved cache locality and reduced cache misses.

In conclusion, the choice of sorting algorithm depends on the order and size of the input data and the tradeoff between time and space complexity. Quicksort with median-of-three pivot selection is a good general-purpose algorithm that performs well on average, while Natural Merge Sort is a good choice when the input data is already partially sorted. However, in some cases, such as when the input data is in reverse order, other algorithms may be more suitable. By carefully considering the characteristics of the input data and the requirements of the problem at hand, we can choose the best sorting algorithm for our particular application.

Sorting Algorithm	Number of Comparisons (Size50)	Number of Exchanges (Size50)
quicksort_with_count random	221	127
quicksort_insertion_sort_100_iterative random	49	1225
quicksort_insertion_sort_50_iterative random	49	1225
quicksort_median_of_three_iterative random	79	355
natural_merge_sort_iterative random	194	632
quicksort_with_count reverse	849	457
quicksort_insertion_sort_100_iterative reverse	49	1225
quicksort_insertion_sort_50_iterative reverse	49	1225
quicksort_median_of_three_iterative reverse	55	258
natural_merge_sort_iterative reverse	133	1225
quicksort_with_count ascending	404	331
quicksort_insertion_sort_100_iterative ascending	49	1225
quicksort_insertion_sort_50_iterative ascending	49	1225
quicksort_median_of_three_iterative ascending	31	255
natural_merge_sort_iterative ascending	0	0


Algorithm Runtimes Comparison
QS-count Random
QS-100it Random
QS-50it Random
QS-median Random
NM-iter Random
QS-count Reversed
QS-100it Reversed
QS-50it Reversed
QS-median Reversed
NM-iter Reversed
QS-count Ascending
QS-100it Ascending
QS-50it Ascending
QS-median Ascending
NM-iter Ascending
Runtime (seconds)
File Input Size
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
0
2000
4000
6000
8000
10000