Test Data for Sorting Algorithm (Practical 9)

1st Data

Number of Integers to sort: 50

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Merge sort | Quick Sort | Median of three (quicksort) | 3-way quicksort |  |
| Mean time (microseconds) | 26.67 | 23.67 | 20.33 | 67.67 | Ascending |
|  | 6.33 | 4.0 | 97.33 | 5.67 | Random |
|  | 2.67 | 3.0 | 1.33 | 5.67 | Descending |
|  | 3.33 | 3.0 | 2.0 | 4.67 | Nearly sorted |
|  | 4.0 | 3.0 | 3.0 | 2.33 | Repeat values |

2nd Data

Number of Integers to sort: 100

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Merge sort | Quick Sort | Median of three (quicksort) | 3-way quicksort |  |
| Mean time (microseconds) | 50.67 | 69.0 | 42.33 | 120.67 | Ascending |
|  | 13.0 | 6.67 | 13.0 | 7.67 | Random |
|  | 5.67 | 13.0 | 3.0 | 27.33 | Descending |
|  | 10.67 | 17.67 | 4.67 | 12.0 | Nearly sorted |
|  | 8.0 | 7.0 | 7.67 | 6.0 | Repeat values |

3rd Data

Number of Integers to sort: 150

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Merge sort | Quick Sort | Median of three (quicksort) | 3-way quicksort |  |
| Mean time (microseconds) | 83.33 | 105.33 | 56.33 | 118.67 | Ascending |
|  | 14.0 | 19.67 | 12.67 | 12.0 | Random |
|  | 9.33 | 65.33 | 17.0 | 47.33 | Descending |
|  | 16.67 | 53.0 | 19.0 | 13.0 | Nearly sorted |
|  | 56.0 | 12.0 | 12.0 | 9.33 | Repeat values |

4th Data

Number of Integers to sort: 500

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Merge sort | Quick Sort | Median of three (quicksort) | 3-way quicksort |  |
| Mean time (microseconds) | 99.33 | 263.33 | 106.33 | 431.33 | Ascending |
|  | 81.67 | 34.33 | 35.33 | 70.0 | Random |
|  | 34.0 | 48.33 | 31.0 | 159.33 | Descending |
|  | 161.67 | 21.0 | 12.67 | 67.33 | Nearly sorted |
|  | 59.67 | 22.67 | 29.67 | 47.0 | Repeat values |

Results Discussion:

**Time Complexity of Merge Sort**

For an array of N elements, from the algorithm written in the Sorts.java file, it will divide the array into two parts, which is the left half (lower than the pivot/middle element) and the right half (larger than the middle element). In this case, it calls itself but only passing half of the elements in the array which will be T(N/2) + T(N/2), resulting to 2T(N/2). At the final part of the algorithm, it merges both the lower and upper half of the array into one single array. Hence, this means that it will take N steps to merge the arrays, and therefore T(N). By adding them together,

In merge sort, the method will keep calling itself until all elements exists by itself in its array. Hence, for every call, it will be

Assuming that N is some power of 2, in this case . By substituting this value into the equation above, it will be

Ignoring the 1 in the equation, the best-case complexity for merge sort algorithm is

In average and worst case, the time complexity remains the same as no matter how many elements are passed in or in whatever order, the division of array will always take place in the middle (N/2) for every call.

**Time Complexity for Quicksort**

Assuming that the partitioning of the array will always be in the middle, this will follow the same principle as merge sort and best-case and average time complexity will similarly be . In the case of O(N), in merge sort, it is the time for merging, while in quicksort, it is the time for partitioning. On the other hand, the worst case can happen by chance or intentionally, if the values in the set are already in ascending order (already sorted!). The reason for this is because instead of having the partition be in the middle. Each time the method is called, it will always partition in the beginning or the array (the first index). An illustration can be seen below (Given an example of 8 sorted elements being passed)

2,8

1,8

Therefore, instead of having N/2

6,8

7,8

8,8

5,8

4,8

3,8

passes, it becomes N passes. And this

results in time complexity to be

From the above data collected by running the SortsTestHarness.java program, we can see that 3-way quicksort algorithm (Dutch National Flag algorithm) can be quite slow especially when the dataset is large and has little to no repeating values. However, can sort data in a flash when repeated values exist in the dataset. In addition to that, when data is in ascending order, it also increase the run time for sorting. Moreover, it is obvious that for all the quicksort algorithm, the longest time taken is only when dataset is already sorted. As mentioned in the time complexity part above, the worst case can go up to . Which is due to the partitioning part where it is not partitioned as close to the middle element as possible.

Merge sort, on the other hand, for all cases (best-average and worst), is known to be equal which is , however may take longer time each time the dataset increases. Hence, in this case merge sort may be more suitable for smaller dataset when compared to quicksort.