Sort Comparisons in C++

This report analyzes the differences between the recursive merge sort and iterative merge sort using the quick sort and heap sort as points of reference. Arrays of size 10; 100; 1,000; 10,000; and 100,000 were constructed and filled with random integers. It was sorted in multiple ways (the array was recreated the same way each time). This following graph and table show the results of the test:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Array Size | Recursive Merge  (microseconds) | Iterative Merge  (microseconds) | Heap  (microseconds) | Quick (augmented with insertion)  (microseconds) |
| 10 | 34 | 9 | 5 | 2\* |
| 100 | 71 | 19 | 18 | 21\* |
| 1000 | 244 | 107 | 149 | 111 |
| 10000 | 3899 | 1220 | 1783 | 1426 |
| 100000 | 26452 | 14162 | 22273 | 18327 |

\*: The insertion sort was used on these to improve performance time at lower array sizes.

Sorts take exponentially larger to perform their functions when the list of elements they have to sort grows. Although the growth increased by factor of ten each time, the length of time taken to perform the sorts, even by the fastest sort, was greater. To prove this point (and to provide a better view of the first four sets of data), the following chart was created which sets aside the arrays of size 100,000 for now.

The recursive merge sort consistently takes the longest time to perform its function despite having a big O notation of O(n log(n)) which is the same average algorithm complexity shared by all of the other sorts presented on this graph. The reason for the differences in performance time between the sorts is not the complexity but the implementation of each sort. The recursive merge sort takes longer since it has so many recursive calls. Although the quick sort uses recursion as well, it is a little more efficient. It is also enhanced by an insertion sort when the original array size is at 100, speeding up the sorting at lower numbers. The iterative version of the merge sort removes all the recursive function calls allowing it to work faster. The heap sort competes very well with the quick but is better off in the worst case performance (since the quick sort goes to O(n2) at worst case). It works by creating a “heap” within the array to locate the largest value to locate next into the sorted portion of the array.

Overall, the recursive merge sort is the slowest sort examined here. This is due to its recursive nature and since it is not enhanced in any way. The heap sort does really well since it works by looking at the array as a binary tree. Although the quick sort is enhanced with the insertion sort, it is still slower at 100 (though faster at 10) than heap. However, it runs faster at larger numbers. The heap sort retains usefulness due to its efficiency even in worst cases. The iterative merge sort fixes the recursive issue and so is able to work faster than any of the other sorts at values of 1,000 and more.

Since most sorts are used for larger arrays, the iterative merge is the recommended for those situations. When the number of elements to sort is at 10,000 the difference between the iterative merge, heap, and quick sorts are very similar (the range between them is 1783 – 1220 = 563 microseconds) whereas the recursive merge is already far slower, with a difference of 2679 microseconds from the best sort. However, this difference rises when the array size scales up to larger numbers leaving the iterative merge sort in the clear lead. The quick sort is not as fast at very large numbers (though comparable still when the array size is at around 10,000) and has a bad worst case scenario. The heap sort is a little worse than the quick but maintains a good worst case scenario complexity of O (n log (n)).