Mariia Romaniuk
November 18, 2019

Summary - Sorting Algorithms Analysis

This program compares the relative performance of different sorting algorithms on datasets containing integers. Ultimately, the data sorted in ascending order. Program tested on two different array sizes: 100 and 1000.

The following random number distributions where used:
Random All – Completely random numbers
Almost Sorted – Almost sorted in ascending order (90% in increasing order, 10% random)
Almost Reversed – Almost sorted in descending order (90% in increasing order, 10% random)
Random Last – Array is sorted except of last 10%

The following sorting algorithms are used:

Selection Sort

Selection Sort is consistent with all cases - $\mathcal{O}(n^2)$ regardless of different datasets. The number of comparisons is always $n(n-1)/2$ and the number of swaps is always $n-1$.

Insertion Sort

InsertionSort is in the worst case scenario $\mathcal{O}(n^2)$ which happens when the list is in reverse order. Insertion is better with the array already sorted but will not perform as good when the array is in descending order as it compares and shifts each value it goes through.

Heap Sort

HeapSort's number of comparison differs slightly depending on how the data is sorted; however, it is $\mathcal{O}(n \log_2 n)$ and does not vary much amongst different data and is not space-inefficient.

Merge Sort

MergeSort is always $\mathcal{O}(n \log_2 n)$ regardless of the order the data is stored. MergeSort is space-inefficient in array implementation. The algorithm requires a temporary array that is the size of the original array to be created to transfer over the sorted data.

Quick Sort

QuickSort is at worst $\mathcal{O}(n^2)$ when the data is almost sorted or almost reversed because it splits the list unbalanced; on the other hand, when the data is sorted in such a way that it divides the list evenly in half every recursive call, it is $\mathcal{O}(n \log_2 n)$. For this reason, Quick Sort does much more efficient on random data than sorted data.

Testing Array of 100 Elements (calculated on an average of three trials per experiment)

Sort Algorithm	*Random All*	*Almost Sorted*	*Almost Reversed*	*Random Last*
Selection Sort	99 swaps 4950 comps	99 swaps 4950 comps	99 swaps 4950 comps	99 swaps 4950 comps
Insertion Sort	2341 swaps 2435 comps	368 swaps 467 comps	4698 swaps 4788 comps	517 swaps 616 comps
Heap Sort	566 swaps 1232 comps	634 swaps 1368 comps	523 swaps 1146 comps	621 swaps 1342 comps
Merge Sort	672 swaps 1203 comps	672 swaps 1111 comps	672 swaps 1073 comps	672 swaps 1049 comps
Quick Sort	339 swaps 639 comps	208 swaps 1837 comps	919 swaps 1843 comps	226 swaps 2126 comps

Testing Array of 1000 Elements (calculated on an average of three trials per experiment)

Sort Algorithm	*Random All*	*Almost Sorted*	*Almost Reversed*	*Random Last*
Selection Sort	999 swaps 499500 comps	999 swaps 499500 comps	999 swaps 499500 comps	999 swaps 499500 comps
Insertion Sort	243034 swaps 244028 comps	29496 swaps 30495 comps	473555 swaps 474508 comps	87562 swaps 88561 comps
Heap Sort	9034 swaps 19068 comps	9552 swaps 20104 comps	8410 swaps 17820 comps	9462 swaps 19924 comps
Merge Sort	9976 swaps 18656 comps	9976 swaps 17414 comps	9976 swaps 17501 comps	9976 swaps 15513 comps
Quick Sort	4700 swaps 13496 comps	5916 swaps 38221 comps	9484 swaps 17420 comps	38158 swaps 48012 comps