Zakaria Almardaee

Assignment 2: Report

Input file	**HalfHeapSort**	**HalfSelection Sort**	**InPlace MergeSort**	**MergeSort**	**QuickSelect**	**StandardSort**
input1.txt	0ms	3000ms	0ms	0ms	0ms	0ms
input2.txt	0ms	3000ms	0ms	0ms	0ms	0ms
input3.txt	0ms	3001ms	0ms	0ms	0ms	0ms
input4.txt	5ms	2892323ms	13ms	18ms	1ms	6ms
input5.txt	5ms	2849652ms	13ms	19ms	1ms	6ms
input6.txt	5ms	2853693ms	12ms	19ms	2ms	6ms
input7.txt	186ms	N/A	428ms	629ms	47ms	234ms
input8.txt	185ms	N/A	429ms	616ms	43ms	230ms
input9.txt	182ms	N/A	433ms	610ms	48ms	230ms
Input Size and Average Time	**1k:** 0ms **31k:** 5ms **1M:** 182ms	**1k:** 3000ms **31k:** 2865214ms **1M:** N/A	**1k:** 0ms **31k:** 13ms **1M:** 430ms	**1k:** 0ms **31k:** 18ms **1M:** 620ms	**1k:** 0ms **31k:** 1ms **1M:** 45ms	**1k:** 0ms **31k:** 6ms **1M:** 320ms

	WorstCaseQuickSelect
Input Size and Average Time	**20k**: 1ms

Algorithm analysis:

- HalfHeapSort algorithm had a big-O of: O(n log n).
    - The worst case is O(n) time to build heap. Since heapify takes O(log n) the overall complexity becomes O(n log n).
- HalfSelectionSort: O(n^2)
    - Has a time complexity of O(n^2) due to it involving nested loops.
- InPlaceMergeSort: O(n log n)
    - Has a time complexity of O(n log n) since it recurcively divides and then merges arrays into a sorted order.
- MergeSort: O(n log n)
    - Has a time complexity of O(n log n) since it recurcively divides and then merges arrays into a sorted order. MergeSort creates extra memory for its merging process.
- QuickSelect: O(n)
    - Has an avgerage time complexity of O(n) when the pivot selection reasonably balances the partitions.
- StandardSort: O(n log n)
    - Has a time complexity of O(n log n) since it uses efficient sorting algorithm such as introsort.
- WorstCaseQuickSelect: O(n^2)
    - Has a time complexity of O(n^2) since the pivot selection leads to uneven partitions consistently.

When comparing the algorithms, we see a clear performance gap, especially between StandardSort and HalfSelectionSort. This difference grows with larger data sets, highlighting that some algorithms, like HalfSelectionSort, fall behind when scaling up. For instance, on 1,000 items, StandardSort and HalfHeapSort are about 3,000 milliseconds quicker than HalfSelectionSort.

Actual test results confirm what we'd expect: less efficient algorithms like HalfSelectionSort lag behind those designed for performance, such as StandardSort and HalfHeapSort. Ranking their speed, QuickSelect comes out on top, followed by HalfHeapSort, StandardSort, InPlaceMergeSort, and MergeSort, with HalfSelectionSort trailing. Notably, HalfHeapSort and StandardSort have a negligible difference on 31,000 items, but at 1 million items, the gap widens to 137 milliseconds, which is worth noting given their similar theoretical complexities.