Patrick Phillips

Class ID: 103

Lab Partner: Brandon Toops

3/28/18

**Tables for Each Array:**

-The top entry of each column is the number of integers input into the sorting algorithm

-The sorting algorithms with O(n^2) average runtimes were not run for 1Mints.txt because it would take a very long time

-Runtimes are all in milliseconds

Array A Data:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Sort: | 1000 | 2000 | 4000 | 8000 | 16000 | 32000 | 1000000 |
| Arrays.Sort() | 2 | 3 | 1 | 0 | 7 | 16 | 375 |
| Bubble Sort | 42 | 82 | 169 | 609 | 2277 | 9488 |  |
| Selection Sort | 23 | 47 | 77 | 278 | 1028 | 3934 |  |
| Insertion Sort | 20 | 37 | 61 | 72 | 187 | 635 |  |
| Mergesort | 4 | 9 | 6 | 12 | 22 | 44 | 833 |
| Quicksort | 4 | 6 | 7 | 16 | 21 | 47 | 1016 |

Array B Data:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Sort: | 1000 | 2000 | 4000 | 8000 | 16000 | 32000 | 1000000 |
| Arrays.Sort() | 0 | 0 | 1 | 0 | 3 | 0 | 82 |
| Bubble Sort | 5 | 21 | 65 | 248 | 987 | 3994 |  |
| Selection Sort | 4 | 16 | 62 | 254 | 992 | 2532 |  |
| Insertion Sort | 0 | 0 | 0 | 0 | 1 | 0 |  |
| Mergesort | 1 | 2 | 3 | 5 | 8 | 18 | 407 |
| Quicksort | 1 | 2 | 2 | 3 | 15 | 0 | 390 |

Array C Data:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Sort: | 1000 | 2000 | 4000 | 8000 | 16000 | 32000 | 1000000 |
| Arrays.Sort() | 1 | 1 | 1 | 0 | 4 | 0 | 31 |
| Bubble Sort | 8 | 27 | 48 | 164 | 635 | 2532 |  |
| Selection Sort | 4 | 17 | 45 | 125 | 550 | 2297 |  |
| Insertion Sort | 2 | 36 | 48 | 126 | 360 | 1278 |  |
| Mergesort | 1 | 2 | 3 | 7 | 7 | 19 | 397 |
| Quicksort | 0 | 1 | 2 | 5 | 4 | 16 | 359 |

Array D Data:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Sort: | 1000 | 2000 | 4000 | 8000 | 16000 | 32000 | 1000000 |
| Arrays.Sort() | 0 | 1 | 2 | 0 | 9 | 31 | 218 |
| Bubble Sort | 6 | 14 | 49 | 218 | 907 | 3631 |  |
| Selection Sort | 4 | 8 | 32 | 250 | 984 | 3918 |  |
| Insertion Sort | 1 | 1 | 3 | 15 | 55 | 245 |  |
| Mergesort | 1 | 1 | 4 | 5 | 9 | 14 | 484 |
| Quicksort | 1 | 1 | 3 | 4 | 6 | 16 | 526 |

**Graphs for each Array:**

**Findings:**

Based on the table and graph of Array A - the unsorted array - the bubble sort has the fastest growing runtime followed by selection sort (the gray line) and insertion sort. They Arrays.Sort() method is the most efficient for Array A, and the mergesort and quicksort have similar runtimes which should scale at at nlog(n) so are much faster then the other 3 sort types.

Based on the table and graph of Array B – the sorted array – the bubble sort takes the longest, followed by the selection sort. The insertion sort takes virtually no time to run because the array is already sorted so it runs in constant time. The mergesort and quicksort and Arrays.Sort() are the next fastest and all have similarly small runtimes. Since this implementation of the quicksort used a random pivot instead of taking the last element, its runtime can be estimated as nlog(n) still instead of n^2.

Based on the table and graph of Array C-the reverse sorted array – the insertion, bubble, and selection sort are the slowest sorts and all have very similar runtimes with the insertion sort being slightly faster than the other two. The mergesort and quicksort then take the next longest times to run, again with very similar values. The arrays.sort() method still is the most efficient.

Based on the table and graph of Array D - the mostly sorted array – the the insertion, bubble, and selection sort are again the slowest sorts and all have very similar runtimes. Insertion sort is faster than the other two by a larger amount however. The mergesort and quicksort take the next longest times to run, with very similar values yet again. The arrays.sort() method is still the most efficient.

**Screenshots:**













