A logo with a flame and stars

Description automatically generatedA logo with a flame and stars

Description automatically generated**Sorting Algorithm Analysis Report**

**Name:** UMUT UYGUR **Section:** 01

**ID:** 13474078970 **Assignment Number:** 2

I developed ways for making random, descending, and ascending arrays in order to complete challenge 2 first. The main method will define the arrays' sizes, which are made up of integers. Since they are private static methods, only members of the class can access them.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| SORT 2 | 10000 | 50000 | 100000 | 500000 |
| ASCARRAY | 1 ms | 15 ms | 16 ms | 110 ms |
| DESCARRAY | 0 ms | 0 ms | 0 ms | 173 ms |
| RANDARRAY | 0 ms | 17 ms | 33 ms | 78 ms |

I then added the provided jar file to the IDE as a referenced library. I then developed an additional technique to arrange the arrays in accordance with the jar file. The method is composed of a millisecond time measurement statement and a switch statement for the sorting algorithms in the jar file. Additionally, I called every sorting technique I could find and recorded how long each one took in the main method. I created a table with all of the information I collected

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| SORT 1 | 10000 | 50000 | 100000 | 500000 |
| ASCARRAY | 0 ms | 0 ms | 16 ms | 16 ms |
| DESCARRAY | 160 ms | 148 ms | 10722 ms | 813959 ms |
| RANDARRAY | 63ms | 2686 ms | 13573 ms | 400450 ms |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| SORT 3 | 10000 | 50000 | 100000 | 500000 |
| ASCARRAY | 0 ms | 15 ms | 33 ms | 188 ms |
| DESCARRAY | 0 ms | 0 ms | 24 ms | 156 ms |
| RANDARRAY | 0 ms | 17 ms | 62 ms | 142 ms |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| SORT 4 | 10000 | 50000 | 100000 | 500000 |
| ASCARRAY | 0 ms | 0 ms | 5 ms | 31 ms |
| DESCARRAY | 235 ms | 158 ms | 17018 ms | 1519030 ms |
| RANDARRAY | 142 ms | 5823 ms | 35734 ms | 1093239 ms |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| SORT 5 | 10000 | 50000 | 100000 | 500000 |
| ASCARRAY | 31 ms | 631 ms | 4146 ms | 91076 ms |
| DESCARRAY | 63 ms | 31 ms | 4944 ms | 108306 ms |
| RANDARRAY | 48 ms | 870 ms | 4321 ms | 154408 ms |

By comparing the outcomes of my execution time measurement, I was able to determine these figures. They are relatively close to the necessary numbers, but they are not the actual values of n², n log n, or linear time.

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithm | Best-Case | Worst-Case | Average-Case |
| Sort1 | O(n) | O( n**2**) | O( n**2**) |
| Sort2 | O( n**2**) | O( n**2**) | O( n**2**) |
| Sort3 | O(n) | O( n**2**) | O( n**2**) |
| Sort4 | O(nlogn) | O( n**2**) | O(nlogn) |
| Sort5 | O(nlogn) | O(nlogn) | O(nlogn) |
| Ascending array is the best-case scenario, descending array is the worst-case scenario and random array is an average case scenario. | | | |

**My Conclusion**

**Sort1**: Likely **Bubble Sort** (O(n**2**) worst-case).

**Sort2**: Likely **Selection Sort** (O(n**2**).

**Sort3**: Likely **Insertion Sort** (O(n**2**) worst-case).

**Sort4**: Likely **Quick Sort** (O(nlogn) average-case, O(n**2**) best-case).

**Sort5**: Likely **Merge Sort** (O(nlogn)).

ANALYSİNG THE DATA PART:

Sort1:

The result indicates that sort1 is bubble sort. Since bubble sort necessitates, at most, n iterations across the entire array, where n is the number of entries in the array. It always looks at the nearby items' order and switches them if needed. The sum of all the swaps and comparisons is therefore about equal to n². In the average-case situation, bubble sort still needs n iterations over the full array, which leads to O(n²). In the best-case situation, bubble sort just needs to go over the array once to ensure that no swaps are required if it is already sorted. As a result, it will do no swaps and only compare. Bubble sort's best-case time complexity is therefore O(n).

Sort2

Sort2 is always a selection sort since it is O(n²). Because, in the worst situation, selection sort necessitates exchanging the current position with the least (or maximum) element through nested loops. The inner loop iterates n times to discover the smallest element for each iteration (where n is the number of elements), which may be shortened to about n². The outer loop iterates n times. The worst-case time complexity for selection sorting is therefore O(n²). On average, selection sort in the average-case scenario necessitates O(n^2) comparisons and swaps due to the requirement for nested loops. Selection sort still needs nested loops to check and determine the minimum (or maximum) element for each place, even in the best-case situation where the array is already sorted. O(n²) time complexity is the outcome of the same amount of comparisons and swaps as in the worst-case scenario.

Sort3

The result indicates that sort3 is an insertion sort. Since there are n items in the array, inserting sort necessitates, at most, going through the entire array n times. For every iteration, up to i comparisons and swaps may need to be performed, where i is the index of the element being put at this time. Since the number of swaps and comparisons in each iteration grows linearly with the size of the sorted component of the array, the worst case scenario has around n × n = n² comparisons and swaps. The worst-case time complexity for insertion sort is therefore O(n²). Likewise, insertion sort still necessitates nested loops in the average-case scenario, which means that on average, O(n²) comparisons and swaps are performed. Insertion sort just needs to go over the array once to ensure that no items need to be relocated, which is ideal when the array is already sorted. As a result, it makes no swaps and only compares. Because of this, the insertion sort's best-case time complexity is O(n).

Sort4

Sort4 is a rapid sort, based on the results. because, in the best-case situation, the partitioning algorithm constantly divides the array in half. With each recursive call, the array is split in half, producing a balanced partitioning. The best-case temporal complexity is therefore O (n log n). On average, Quick Sort's temporal complexity is O (n log n). The array is divided into two sections via the partitioning method, which requires linear time for each operation.

Sort5

The result indicates that sort5 is a merge sort. Because Merge Sort works just as well in the best-case situation as it does in the average-case scenario. Until each individual element is reached, it continuously splits the array in half before recombining them in a sorted manner. The size of the merged subarrays determines how long the merging process takes, which is linear. Merge sort is therefore always O (n log n).