**Exploring Sorting Runtimes**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Array Size | Sort Type | Bubble Sort | Selection Sort | Insertion Sort |
| 10 | Ascending | 1.0 | 1.0 | 0.0 |
|  | Descending | 2.0 | 1.0 | 1.0 |
|  | Random | 2.6 | 2.3 | 2.7 |
|  | Nearly Sorted | 9.3 | 1.0 | 0.3 |
| 100 | Ascending | 754.6 | 453.3 | 3.0 |
|  | Descending | 202.3 | 95.3 | 127.7 |
|  | Random | 431.6 | 98.7 | 65.0 |
|  | Nearly Sorted | 817.3 | 199.3 | 13.0 |
| 500 | Ascending | 463.0 | 995.6 | 23.3 |
|  | Descending | 568.3 | 1063.0 | 615.3 |
|  | Random | 573.3 | 666.3 | 411.3 |
|  | Nearly Sorted | 1037.3 | 622.0 | 229.6 |
| 1000 | Ascending | 2615.7 | 3036.7 | 29.7 |
|  | Descending | 2295.3 | 3200.0 | 2906.7 |
|  | Random | 2447.7 | 2296.3 | 4815.7 |
|  | Nearly Sorted | 2182.7 | 3636.7 | 459.3 |

|  |  |  |  |
| --- | --- | --- | --- |
| Array Size | Sort Type | Quick Sort | Merge Sort |
| 10 | Ascending | 1.3 | 3.3 |
|  | Descending | 2.0 | 3.0 |
|  | Random | 1.7 | 9.7 |
|  | Nearly Sorted | 1.3 | 3.7 |
| 100 | Ascending | 156.0 | 79.3 |
|  | Descending | 120.7 | 89.0 |
|  | Random | 56.3 | 73.7 |
|  | Nearly Sorted | 87.3 | 90.0 |
| 500 | Ascending | 573.3 | 212.3 |
|  | Descending | 2210.0 | 305.7 |
|  | Random | 67.7 | 150.0 |
|  | Nearly Sorted | 116.3 | 204.3 |
| 1000 | Ascending | 1496.7 | 163.7 |
|  | Descending | 1927.7 | 230.7 |
|  | Random | 1716.7 | 365.3 |
|  | Nearly Sorted | 431.3 | 680.7 |

Runtimes for Three QuickSort Implementation and QuickSort Implementation.

|  |  |  |  |
| --- | --- | --- | --- |
| Array Size | Sort Type | quicksortmedian3 | quicksortrandom |
| 10 | Ascending | 1.3 | 4.3 |
|  | Descending | 1.3 | 3.7 |
|  | Random | 5.0 | 3.7 |
|  | Nearly Sorted | 1.7 | 25.0 |
| 100 | Ascending | 33.0 | 119.3 |
|  | Descending | 42.7 | 87.3 |
|  | Random | 81.3 | 86.0 |
|  | Nearly Sorted | 87.3 | 343.0 |
| 500 | Ascending | 132.0 | 354.3 |
|  | Descending | 126.0 | 318.7 |
|  | Random | 105.7 | 68.3 |
|  | Nearly Sorted | 95.0 | 437.3 |
| 1000 | Ascending | 212.0 | 635.7 |
|  | Descending | 174.7 | 321.3 |
|  | Random | 122.7 | 123.0 |
|  | Nearly Sorted | 323.3 | 526.0 |

The sorting algorithms used in the Sorts class have various time complexities and unique properties:

Bubble sort - The average and worst case time complexity of bubble sort is O(n2). When two neighboring components are in the wrong sequence, it constantly compares them and switches them. Although bubble sort is easy to use, it is ineffective for huge datasets.

Selection sort - The average and worst case time complexity of selection sort is O(n2). It separates the array into a sorted and an unsorted zone, then swaps the initial unsorted member with the minimum element from the unsorted region periodically. In terms of the quantity of swaps, selection sort outperforms bubble sort, although it still has a lot of comparisons.

Insertion sort - In both the average and worst instances, the time complexity of insertion sort is O(n2). The sorted section is gradually built by inserting each element into its proper position among the already sorted items as it iterates through the array. For tiny or almost sorted arrays, insertion sort works effectively; but, for big or reverse-sorted arrays, it is ineffective.

Merge sort - In both the average and worst scenarios, the time complexity of merge sort is O(n log n). Recursively splitting the array into smaller subarrays, sorting them, and then merging them back together is how the divide-and-conquer strategy is used. Large datasets are a good fit for merge sort because of their strong scalability.

Quicksort - Quicksort's time complexity is O(n log n) in the average case and O(n2) in the worst situation, however with suitable pivot selection methods, the worst-case scenario is unlikely to happen. All elements smaller than the pivot are positioned to the left of the pivot, and all elements larger than the pivot are positioned to the right. This is how Quicksort divides the array. The subarrays on either side of the pivot are then sorted iteratively. Because of its effective partitioning, Quicksort is frequently faster than other sorting algorithms.

Quicksortmedian3 - By employing the median value between the left, center, and right elements, the quicksort variation with median-of-three partitioning enhances the pivot selection. This lowers the likelihood of selecting a poor pivot and enhances quicksort performance.

Quicksortrandom - A random element is chosen as the pivot in the quicksort version with random partitioning, helping to prevent worst-case scenarios and ensuring good average performance. Quicksort becomes more resistant to hostile input by adding a randomness component through random partitioning.

In conclusion, there are various temporal complexities and properties among the sorting algorithms. While easy, bubble sort and selection sort are ineffective for large datasets. For tiny or virtually sorted arrays, insertion sort performs well. Large datasets are best suited for merge sort and quicksort because of their improved average-case performance. The performance of the technique is improved by quicksort variants that use random and median-of-three partitioning to improve pivot selection. The sorting algorithm selected is determined by the application's particular needs as well as the qualities of the input data.