|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Sorting Algorithm | List | Number of Elements | Number of Swaps | Number of Comparisons |
| Selection Sort | Inverse order | 100 | 99 | 4950 |
| Inverse order | 1000 | 999 | 499500 |
| Almost sorted | 100 | 99 | 4950 |
| Almost sorted | 1000 | 999 | 499500 |
| Random | 100 | 99 | 4950 |
| Random | 1000 | 999 | 499500 |
| Insertion Sort | Inverse order | 100 | 4950 | 5049 |
| Inverse order | 1000 | 499500 | 500499 |
| Almost sorted | 100 | 112 | 211 |
| Almost sorted | 1000 | 11538 | 12537 |
| Random | 100 | 2692 | 2791 |
| Random | 1000 | 248222 | 249221 |
| Merge Sort | Inverse order | 100 | 1344 | 672 |
| Inverse order | 1000 | 19952 | 9976 |
| Almost sorted | 100 | 1344 | 672 |
| Almost sorted | 1000 | 19952 | 9976 |
| Random | 100 | 1344 | 672 |
| Random | 1000 | 19952 | 9976 |
| Quick Sort | Inverse order | 100 | 2599 | 4950 |
| Inverse order | 1000 | 250999 | 499500 |
| Almost sorted | 100 | 238 | 2761 |
| Almost sorted | 1000 | 88032 | 148783 |
| Random | 100 | 314 | 658 |
| Random | 1000 | 5503 | 4842 |
| Heap Sort | Inverse order | 100 | 517 | 567 |
| Inverse order | 1000 | 8317 | 8817 |
| Almost sorted | 100 | 636 | 686 |
| Almost sorted | 1000 | 9378 | 9878 |
| Random | 100 | 584 | 675 |
| Random | 1000 | 5407 | 10571 |
| Bubble Sort  (Improved) | Inverse order | 100 | 4950 | 4950 |
| Inverse order | 1000 | 499500 | 499500 |
| Almost sorted | 100 | 110 | 110 |
| Almost sorted | 1000 | 13250 | 13250 |
| Random | 100 | 2424 | 2424 |
| Random | 1000 | 250468 | 250468 |

Student: Inna Yedzinovich

Course: CSC 430 (DATA STRUCTURE II)

Professor: Anna Salvati

Sorting Algorithms Table

Sorting Algorithms Analysis

*Selection sort algorithm* sorts an array by repeatedly finding the minimum element from unsorted part and putting it at the beginning. Time complexity for this algorithm is O(n^2) as there are two nested-loops. In terms of space, this algorithm never goes more them O(n) swaps and it is good for space costly operations. This algorithm is stable.

*Insertion Sort* is comparing the array [ i+1] index of the array with array [i] index until it reaches zero value or array [i+1] is grader then array [i]. Time complexity for insertion sort algorithm is O(n^2), but it could by significantly less for almost sorted array. Insertion sort shows minimum time for fully sorted array. But insertion sort takes maximum time to sort if elements are sorted in reverse order. This algorithm is stable.

*Merge Sort* is a Divide and Conquer algorithm. It divides input array into two equals parts (halves), calls itself for the two halves and then merges the two sorted halves. Merge Sort is a recursive algorithm and can be used either for array or linked list. Time complexity for this sorting algorithm is O(n log n ) is all possible cases. Compare to the Quick Sort, Merge sort performs better on soring array in ascendant/descendant orders. This algorithm is stable.

*Quick Sort* is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. It is a recursive way of soring, but average and best cases for quick sort are O(n log(n)). Worst case is almost O(n^2) for sorted in descendent or ascendant orders arrays. Best case for quick sort is an array with random numbers. This algorithm is unstable.

*Heap Sort* is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. It repeats the same process for remaining element. Time complexity of heapDown is O(Log n). Time complexity of create a heap is O(n) and overall time complexity of Heap Sort is O(n Log n). Compare to Quick or Merge sort, Heap sort is not the best choice. But for almost sorted or sorted in ascendant order list, heap sort may perform better then merge. This algorithm is unstable.

*Bubble Sort (improved)* is a simple sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order. The regular bubble sort function always runs O(n^2) time even if the array is sorted. But it can be optimized by stopping the algorithm if inner loop didn’t cause any swap (see code implementation for bubble sort). Worst case O(n^2) occurs when array is reverse sorted. Best case O(n) occurs when array is already sorted. This algorithm is stable.

Conclusion: Overall, there is no best sorting algorithms that could perform best in all cases. All algorithms perform differently depending on its content.