|  |  |  |
| --- | --- | --- |
| Insertion Sort | | |
| Input Size | **Random  (milliseconds)** | **Almost Sorted (%5)  (milliseconds)** |
| N = 103 | 1st run: 13.81  2nd run: 12.29 **Avg: 13.25**  3rd run: 13.66 | 1st run: 0.12  2nd run: 0.13 **Avg: 0.13**  3rd run: 0.13 |
| N = 104 | 1st run: 71.32  2nd run: 65.99 **Avg: 68.11**  3rd run: 67.03 | 1st run: 1.27  2nd run: 1.36 **Avg: 1.29**  3rd run: 1.25 |
| N = 105 | 1st run: 4505.57  2nd run: 4527.12 **Avg: 4521.04**  3rd run: 4530.42 | 1st run: 6.49  2nd run: 6.59 **Avg: 6.61**  3rd run: 6.76 |
| N = 106 | 1st run: 1872311.03  2nd run: 1870529.45 **Avg: 1866144.78**  3rd run: 1855593.86 | 1st run: 17.72  2nd run: 17.07 **Avg: 16.07**  3rd run: 13.34 |

|  |  |  |
| --- | --- | --- |
| Quick Sort | | |
| Input Size | **Random  (milliseconds)** | **Almost Sorted (%5)  (milliseconds)** |
| N = 103 | 1st run: 1.59  2nd run: 1.79 **Avg: 1.61**  3rd run: 1.47 | 1st run: 1.04  2nd run: 1.40 **Avg: 1.15**  3rd run: 1.01 |
| N = 104 | 1st run: 23.44  2nd run: 21.66 **Avg: 22.43**  3rd run: 22.19 | 1st run: 16.45  2nd run: 18.93 **Avg: 18.42**  3rd run: 19.90 |
| N = 105 | 1st run: 43.40  2nd run: 106.24 **Avg: 94.18**  3rd run: 132.92 | 1st run: 65.36  2nd run: 64.75 **Avg: 68.81**  3rd run: 75.33 |
| N = 106 | 1st run: 253.75  2nd run: 264.09 **Avg: 256.23**  3rd run: 251.87 | 1st run: 98.88  2nd run: 95.47 **Avg: 97.31**  3rd run: 97.58 |

|  |  |  |
| --- | --- | --- |
| Shell Sort (Shell’s sequence) | | |
| Input Size | **Random**  **(milliseconds)** | **Almost Sorted (%5)**  **(milliseconds)** |
| N = 103 | 1st run: 2.64  2nd run: 2.62 **Avg: 2.38**  3rd run: 1.89 | 1st run: 1.06  2nd run: 1.01 **Avg: 1.03**  3rd run: 1.03 |
| N = 104 | 1st run: 23.77  2nd run: 21.28 **Avg: 22.59**  3rd run: 22.71 | 1st run: 10.61  2nd run: 10.01 **Avg: 10.38**  3rd run: 10.53 |
| N = 105 | 1st run: 41.94  2nd run: 41.47 **Avg: 43.40**  3rd run: 46.78 | 1st run: 17.53  2nd run: 17.65 **Avg: 17.54**  3rd run: 17.45 |
| N = 106 | 1st run: 1048.11  2nd run: 1076.43 **Avg:** **1044.10**  3rd run: 1007.77 | 1st run: 82.66  2nd run: 83.88 **Avg: 83.49**  3rd run: 86.64 |

|  |  |  |
| --- | --- | --- |
| Shell Sort (Knuths's increments) | | |
| Input Size | **Random**  **(milliseconds)** | **Almost Sorted (%5)**  **(milliseconds)** |
| N = 103 | 1st run: 1.85  2nd run: 1.62 **Avg: 1.68**  3rd run: 1.58 | 1st run: 0.57  2nd run: 0.64 **Avg: 0.63**  3rd run: 0.70 |
| N = 104 | 1st run: 20.27  2nd run: 20.08 **Avg: 19.91**  3rd run: 19.38 | 1st run: 7.76  2nd run: 8.09 **Avg: 8.04**  3rd run: 8.28 |
| N = 105 | 1st run: 38.63  2nd run: 36.55 **Avg: 37.08**  3rd run: 36.08 | 1st run: 14.36  2nd run: 14.15 **Avg: 14.24**  3rd run: 14.21 |
| N = 106 | 1st run: 779.78  2nd run: 763.95 **Avg: 765.39**  3rd run: 752.44 | 1st run: 52.21  2nd run: 52.28 **Avg: 52.61**  3rd run: 53.34 |

Discussion:

**Insertion Sort**

Compared to the other sorting algorithms, insertion sort is less efficient on large inputs. It iterates through an input list consuming an element during each repetition. Each iteration removes a sorted element and places it in it’s exact location in the sorted list. From our analysis, insertion sort works well on an almost-sorted list. Since 5% of the elements in the almost-sorted array we created were just shifted right once, it could sort them in one pass and at N = 106 it took 16 milliseconds. It would have a O(n) in a best case, when the input is already or almost sorted. The worst case is for large inputs in reverse order and for random inputs, both O(n2). In our runs, the random at N = 106 took approximately 30 mins.

**Shell Sort** works great with almost sorted inputs because it is based on insertion sort; it compares elements far apart, steadily reducing the gap between elements being sorted. it then performs insertion sort on adjacent elements.

* **Donald Shell’s original sequence** repeatedly divides by 2 to determine how far apart elements to be sorted are. At N=106 it takes 1044 milliseconds on average to sort a random list, four times slower than quicksort. Shell’s sequence has a time complexity of O(n2). This would not be a good implementation for a large list of random values, but it would be good to use on a large list of almost-sorted values or when the list is not very large.
* **Knuths’s increments** (1, 4, 13, 40..) has a slightly better performance and a time complexity of O(N3/2). At N = 106 it took 765 milliseconds to sort a random list. This too would be a simple implementation for a list of small values, or almost sorted integers. It works quite well for smaller random and almost-sorted files because it doesn’t exceed 100 milliseconds up to N = 105.

**Quicksort**

Mathematically, quicksort has an average running time of O(NlogN). It works best for random numbers, large inputs, and inputs where less than relation is defined. The worst case for quicksort is O(n2), the rare event when the pivot is the largest or smallest element in the array. Quicksort uses divide-and-conquer to sort the array, it splits the array into two subarrays and recursively sorts them. In our test runs, it performed faster than all the other algorithms for a random list, but slower compared to others for the almost-sorted list on small inputs. This would be the ideal algorithm to use for very large inputs of random data that can be performed in main memory.