|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Amount of  Elements | Generate  List | Insertion  Sort | Quick Sort  First Pivot | Quick Sort Random Pivot |
| 10 | 0.0 | 0.0 | 0.0 | 0.0 |
| 100 | 0.0 | 0.0 | 0.0 | 0.0 |
| 1,000 | 0.0 | 0.02643 | 0.01327 | 0.01245 |
| 10,000 | 0.01264 | 2.89078 | 0.01562 | 0.03124 |
| 50,000 | 0.01550 | 72.44597 | 0.10986 | 0.12796 |
| 100,000 | 0.06771 | 293.90385 | 0.22499 | 0.28796 |
| 1,000,000 | 0.67725 | 41859.90171 | 2.96465 | 3.73214 |

For this setting, please say which algorithm is better, Quick\_sort\_random or Quick\_sort\_first, and say why. When might this be different?

For this setting, based on the stats we have, Quick\_sort\_first method appears to be the best in all those algorithms, because it costs the least time to finish the whole calculation process. Choosing a pivot at random makes it harder to create a data set that generates O(N^2) performance. But it depends on our requirements. For most data, picking the first is sufficient. But, if we find that we're running into worst-case scenarios (like partially sorted input), choosing a random pivot minimizes the chance that we may encounter worst-case O(n2) performance (always choosing first would cause worst-case performance for nearly-sorted or nearly-reverse-sorted data). Quick\_sort's worst case runtime occurs when partitioning results in one array of 1 element, and one array of n-1 elements. When we choose the first element as the partition. And if we have an array that is in decreasing order, the first pivot will be the biggest, so everything else in the array will move to the left of it. Then when we recurse, the first element will be the biggest again, so once more we need to put everything to the left of it, and so on. However, picking any arbitrary element runs the risk of poorly partitioning the array of size n into two arrays of size 1 and n-1. If we do that often enough, our quicksort runs the risk of becoming O(n^2).