|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **n=100** | **n=5000** | **n=100,000** | **n=1000,0000** |
| **Insertion Sort** | 00:00:00.0000176 | 00:00:00.0296039 | 00:00:12.1786702 | N/A |
| **Selection sort** | 00:00:00.0000126 | 00:00:00.0169590 | 00:00:06.50850824 | N/A |
| **Bubble Sort** | 00:00:00.0000529 | 00:00:00.0890737 | 00:00:37.0863713 | N/A |
| **Merge sort** | 00:00:00.0000233 | 00:00:00.0008521 | 00:00:00.0174260 | 00:00:02.2708713 |
| **Quick sort** | 00:00:00.0000086 | 00:00:00.0004395 | 00:00:00.0125653 | 00:00:01.6680562 |
| **Using Lambda** | 00:00:00.0000087 | 00:00:00.0004999 | 00:00:00.0124250 | 00:00:01.6719039 |

From doing all of these timestamps, I have reached a conclusion that the best overall sorting algorithm, at least on my PC and in my case, is Quick sort. Following closely by are merge sort and sorting using lambda expression. The reason why is that they are most optimal for large datasets (n = 1000,0000). They can do it ~2 seconds. Meanwhile, insertion, selection, and bubble sort are really incompatible when it comes to working with large datasets. In the end, the hardware where the code is being ran also affects the results.

In my opinion, avoiding mutations offers a significant benefit, which is that it enhances the quality of the code, as well as making the code easier to debug. The reason for this is that immutable data eliminates any side effects and ensures consistent behaviour. When it comes to the delegates, they enable dynamic behaviour, as well as modularity and code reuse in my case. Since it also allows timing and/or event handling, it was useful in this exercise.