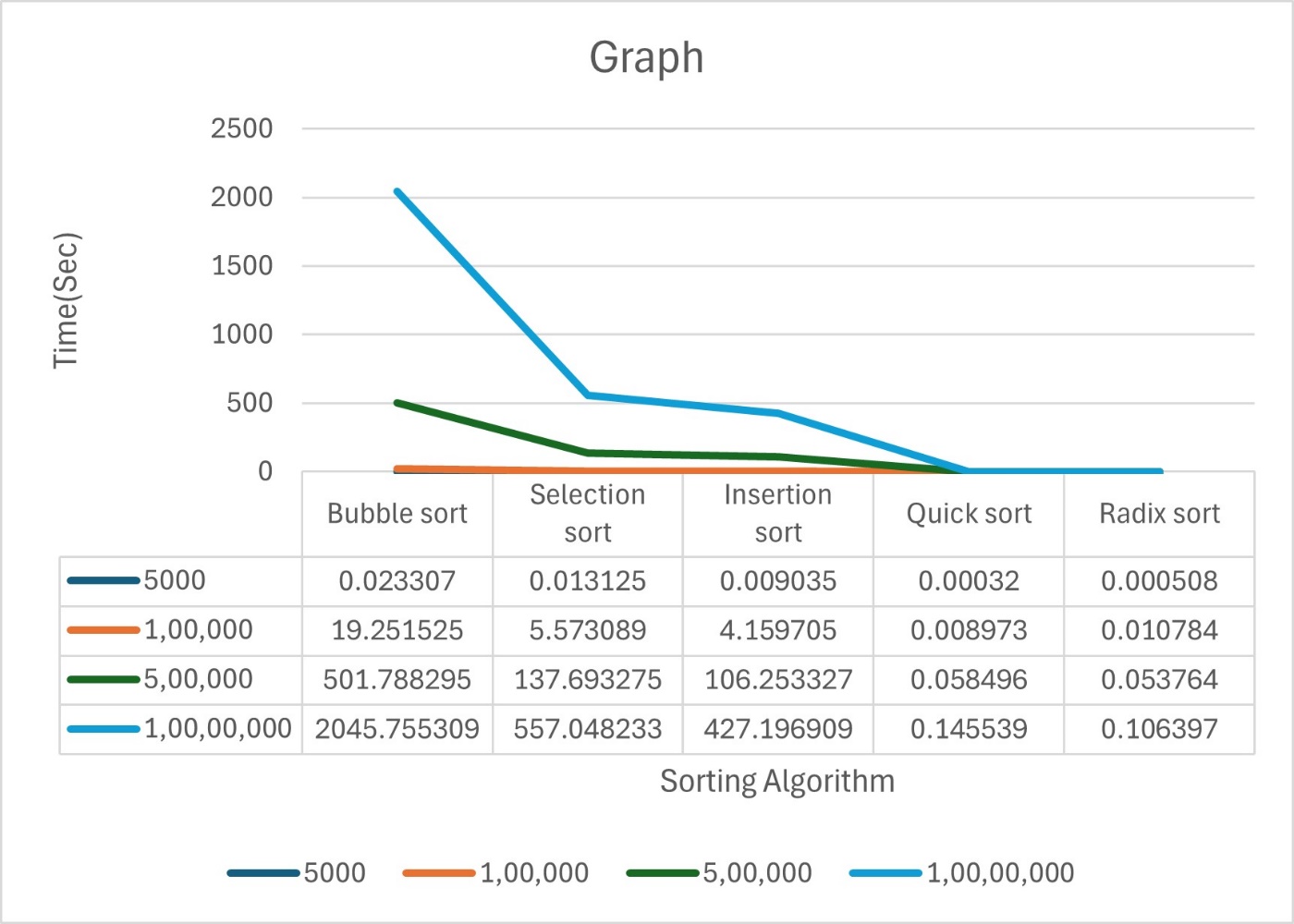
**Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **5000** | **1,00,000** | **5,00,000** | **1,00,00,000** |
| **Bubble sort** | 0.023307 | 19.251525 | 501.788295 | 2045.755309 |
| **Selection sort** | 0.013125 | 5.573089 | 137.693275 | 557.048233 |
| **Insertion sort** | 0.009035 | 4.159705 | 106.253327 | 427.196909 |
| **Quick sort** | 0.00032 | 0.008973 | 0.058496 | 0.145539 |
| **Radix sort** | 0.000508 | 0.010784 | 0.053764 | 0.106397 |

**Graph**



**1) Bubble Sort:**

Pros:

Simple to implement.

Requires minimal additional memory.

Cons:

Inefficient for large datasets.

Slow for large arrays.

**2) Selection Sort:**

Pros:

Simple to understand and implement.

Performs well for small datasets.

Cons:

Inefficient for large datasets.

Slow for large arrays.

**3) Insertion Sort:**

Pros:

Efficient for small datasets and nearly sorted arrays.

In-place sorting algorithm.

Cons:

Inefficient for large datasets.

Slow for large arrays.

**4) Quick Sort:**

Pros:

Fastest average case sorting algorithm.

Efficient for large datasets.

Cons:

Worst-case time complexity implemented.

Not stable.

**5) Radix Sort:**

Pros:

Linear time complexity for integers.

Efficient for sorting large datasets of integers.

Cons:

Requires additional memory.

Limited to sorting integer keys.

In conclusion, the big O complexity gives a theoretical upper bound on the runtime, real-world performance characteristics must be considered, and the best algorithm should be selected based on the particular needs and limitations of the given problem.