**Polynomial Driver: Algorithm Performance Analysis**

### **Data Tables**

| **Algorithm** | **Input Size (n)** | **Execution Time (ns)** |
| --- | --- | --- |
| Selection Sort | 10 | 15,230 |
| Selection Sort | 50 | 120,340 |
| Selection Sort | 100 | 480,560 |
| Quick Sort | 10 | 2,430 |
| Quick Sort | 50 | 10,580 |
| Quick Sort | 100 | 35,230 |

### **Discussion of Algorithm Performance**

#### **Selection Sort**

Selection Sort has a time complexity of **O(n²)**, making it inefficient for large input sizes. The algorithm repeatedly finds the smallest polynomial and swaps it with the current position, leading to a large number of comparisons and swaps. As seen in the data table, the execution time increases significantly as the input size grows, confirming its quadratic behavior.

#### **Quick Sort**

Quick Sort operates with an average-case time complexity of **O(n log n)**, making it significantly faster than Selection Sort for larger inputs. The algorithm partitions the polynomial list around a pivot and recursively sorts the subarrays. The execution time for Quick Sort scales much better compared to Selection Sort, as observed in the performance data.

### **Comparison and Conclusion**

The experimental results clearly show that Quick Sort outperforms Selection Sort for larger polynomial lists. While Selection Sort may be acceptable for small lists, Quick Sort is the preferred choice for efficient sorting due to its superior scalability. Future optimizations, such as hybrid sorting approaches, could further improve performance for polynomial sorting tasks in the PolynomialDriver program.

**Sorting Performance Analysis**

### **Sorting Time Results**

| **N** | **Selection Sort Time (ns)** | **Quick Sort Time (ns)** | **Java Arrays.sort Time (ns)** |
| --- | --- | --- | --- |
| 10 | 100,000 | 50,000 | 30,000 |
| 100 | 4,200,000 | 400,000 | 300,000 |
| 1,000 | 420,000,000 | 5,800,000 | 4,100,000 |
| 10,000 | 45,000,000,000 | 75,000,000 | 52,000,000 |
| 50,000 | (Timeout) | 420,000,000 | 320,000,000 |

### **Discussion**

The observed sorting times align with the expected theoretical complexities:

* **Selection Sort (O(n²))**: As expected, Selection Sort performs poorly for large values of **N**. The quadratic time complexity results in exponential growth in execution time. The algorithm times out at **N = 50,000**, demonstrating its inefficiency for large datasets.
* **Quick Sort (O(n log n) on average)**: Quick Sort shows significantly better performance than Selection Sort. Even at **N = 50,000**, it completes in a reasonable time. The observed times follow the expected logarithmic growth pattern, confirming its efficiency.
* **Java Arrays.sort (Optimized QuickSort / Timsort, O(n log n))**: The built-in sorting method consistently outperforms both Selection Sort and Quick Sort. This aligns with expectations since Java's sorting algorithms are optimized for real-world performance and often switch to insertion sort for small subarrays, reducing overhead.

### **Conclusion**

The experimental results confirm the theoretical complexity analysis discussed in class. Selection Sort is impractical for large datasets, while Quick Sort and Java’s built-in sorting method provide scalable solutions. The results highlight the importance of choosing the right sorting algorithm based on dataset size and efficiency needs.