# Task 1.1 - Written Task for Chosen Sorting Algorithm

## Chosen Sorting Algorithm:

The C# OrderByDescending() LINQ method was utilised in order to sort orders by Delivery Date(newest first). Internally, this depends on Introsort, a combination of InsertionSort, HeapSort, and QuickSort.

## Asymptotic Time Complexities:

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithm | Best Case | Average Case | Worst Case |
| Introsort (OrderByDescending) | O(n log n) | O(n log n) | O(n log n) |
| Merge Sort | O(n log n) | O(n log n) | O(n log n) |
| Quick Sort | O(n log n) | O(n log n) | O(n²) |

## Comparative Evaluation:

- Quick Sort (used in part A):  
 - Works very well in practice, but has a bad worst-case performance.   
 - Not stable (relative order of equal elements may change).  
  
- Merge Sort (used in part B):  
 - Predictable performance (O(n log n) across all cases).  
 - Stable sort, making it ideal for datasets with equal keys.  
 - Requires additional memory due to recursion and array copying.  
  
- Built-in (Introsort) (used in part C):  
 - Combines the best aspects of Quick Sort, Heap Sort, and Insertion Sort.  
 - Maintains O(n log n) even in worst-case scenarios.  
 - Uses native optimizations and avoids recursion pitfalls.

## Conclusion:

The built-in OrderByDescending() (Introsort) is the best of the three for sorting objects in the real world. It provides:  
- Reliable worst-case performance,  
- Good cache locality,  
- And native framework-level optimization and stability.  
  
So, for sorting things like Order by Delivery Date, it's better than Merge or Quick Sort.

# Task 2.2 C – Written Task for PRNG Correctness and Intractability

1. **Is your PRNG implementation correct?**

* Yes, it is correct based on the test output, which confirms
* All generated numbers fall within the specified range [1,1000]
* The sequence is not sorted in any order, indicating randomness

1. **Is your PRNG implementation intractable?**

* No, the PRNG implementation is not intractable.
* The empirical timing results show that generating even 1,000,000 random numbers took only 10 milliseconds, demonstrating that the algorithm is highly efficient and scales well.
* The Log-Log graph from Excel (based on recorded timings) further supports this with near-linear growth.