MINISTRY OF EDUCATION AND RESEARCH OF REPUBLIC OF MOLDOVA
TECHNICAL UNIVERSITY OF MOLDOVA
FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS
DEPARTMENT OF SOFTWARE AND AUTOMATION ENGINEERING

ANALYSIS OF ALGORITHMS

LABORATORY WORK #2

Study and empirical analysis of sorting algorithms

Author:

Andrei Chicu

std. gr. FAF–233

Verified:

Fistic Cristofor

Department of SEA, FCIM UTM

Chișinău, 2025

1 Analysis of Algorithms

github url: `https://github.com/andyp1xe1/aa_labs/tree/main/lab2`

1.1 Objective

The main objective of this laboratory work is to study and empirically analyze various sorting algorithms, particularly focusing on QuickSort, MergeSort, HeapSort, and PatienceSort. Through implementation and testing, we aim to evaluate their performance across different input conditions and establish a comparative framework to understand their strengths and limitations.

1.2 Tasks

1. Implement QuickSort, MergeSort, HeapSort, and PatienceSort algorithms in Python
2. Define various input data properties for testing the algorithms
3. Establish appropriate metrics for algorithm comparison
4. Conduct empirical analysis of all algorithms across different input sizes and types
5. Create graphical representations of the performance results
6. Analyze the results and draw conclusions about algorithm efficiency

1.3 Theoretical Notes

An alternative to mathematical analysis of complexity is empirical analysis. This may be useful for: obtaining preliminary information on the complexity class of an algorithm; comparing the efficiency of two (or more) algorithms for solving the same problems; comparing the efficiency of several implementations of the same algorithm; obtaining information on the efficiency of implementing an algorithm on a particular computer. In the empirical analysis of an algorithm, the following steps are usually followed:

1. The purpose of the analysis is established.
2. Choose the efficiency metric to be used (number of executions of an operation (s) or time

execution of all or part of the algorithm.

1. The properties of the input data in relation to which the analysis is performed are established

(data size or specific properties).

1. The algorithm is implemented in a programming language.
2. Generating multiple sets of input data.
3. Run the program for each input data set.
4. The obtained data are analyzed.

The choice of the efficiency measure depends on the purpose of the analysis. If, for example, the aim is to obtain information on the complexity class or even checking the accuracy of a theoretical estimate then it is appropriate to use the number of operations performed. But if the goal is to assess the behavior of the implementation of an algorithm then execution time is appropriate.

After the execution of the program with the test data, the results are recorded and, for the purpose of the analysis, either synthetic quantities (mean, standard deviation, etc.) are calculated or a graph with appropriate pairs of points (i.e. problem size, efficiency measure) is plotted.

1.4 Introduction

Sorting is a fundamental operation in computer science that arranges elements in a specific order (typically ascending or descending). This laboratory work focuses on four distinct sorting algorithms: QuickSort, MergeSort, HeapSort, PatienceSort. It is known that sorting algorithms perform differently with different kinds of data. Some are more performant on random arrays, while others perform well with almost sorted once as well. This will be tested impirically in this laboratory work.

1.5 Comparison Metric

The comparison metric for this laboratory work will be considered the time of execution of each algorithm (T(n)).

1.6 Input Format

To thoroughly test the sorting algorithms, we will use various input types that represent different real-world scenarios:

1. **Random Arrays**: Arrays with elements in random order, representing the average case for most sorting algorithms.

```
generate_random_array(size)
```

2. **Nearly Sorted Arrays**: Arrays where most elements (90%) are already in their correct positions, with only a few elements out of place.

```
generate_nearly_sorted_array(size, percent_sorted=0.9)
```

3. **Reverse Nearly Sorted Arrays**: Arrays that are nearly sorted in reverse order.

```
generate_reverse_nearly_sorted_array(size, percent_sorted=0.9)
```

4. **Sorted Arrays**: Arrays already in sorted order, representing the best case for many algorithms.

```
generate_sorted_array(size)
```

5. **Reverse Sorted Arrays**: Arrays in descending order, often representing the worst case for certain algorithms.

```
generate_reverse_sorted_array(size)
```

6. **Arrays with Duplicates**: Arrays where only a small percentage (10%) of elements are unique, testing how algorithms handle repeated values.

```
generate_duplicates_array(size, unique_percent=0.1)
```

The input sizes will range from small (10 elements) to large (10,000 elements) to observe how the algorithms scale: `[10, 50, 100, 500, 1000, 2000, 3000, 4000, 5000, 7000, 10000]`.

2 Implementation

2.1 Quick Sort

Quick Sort is a divide-and-conquer algorithm that selects a 'pivot' element from the array and partitions other elements into two sub-arrays according to whether they are less than or greater than the pivot. The sub-arrays are then recursively sorted. The average time complexity is O(n log n), but the worst-case scenario is $O(n^2)$ when the pivot selection is poor (e.g., always selecting the first or last element in a sorted array).

2.1.1 Implementation

```
def partition(arr, low, high):
    pivot = arr[high]

    i = low - 1
    for j in range(low, high):

        if arr[j] < pivot:
            i += 1

            swap(arr, i, j)
    swap(arr, i + 1, high)

    return i + 1


def swap(arr, i, j):

    arr[i], arr[j] = arr[j], arr[i]


def quick_sort_rec(arr, low, high):

    if low < high:
        p_idx = partition(arr, low, high)

        quick_sort_rec(arr, low, p_idx - 1)
        quick_sort_rec(arr, p_idx + 1, high)



def quick_sort(arr):
    quick_sort_rec(arr, 0, len(arr)-1)
```

Listing 1: Implementation of Quick Sort

2.1.2 Results

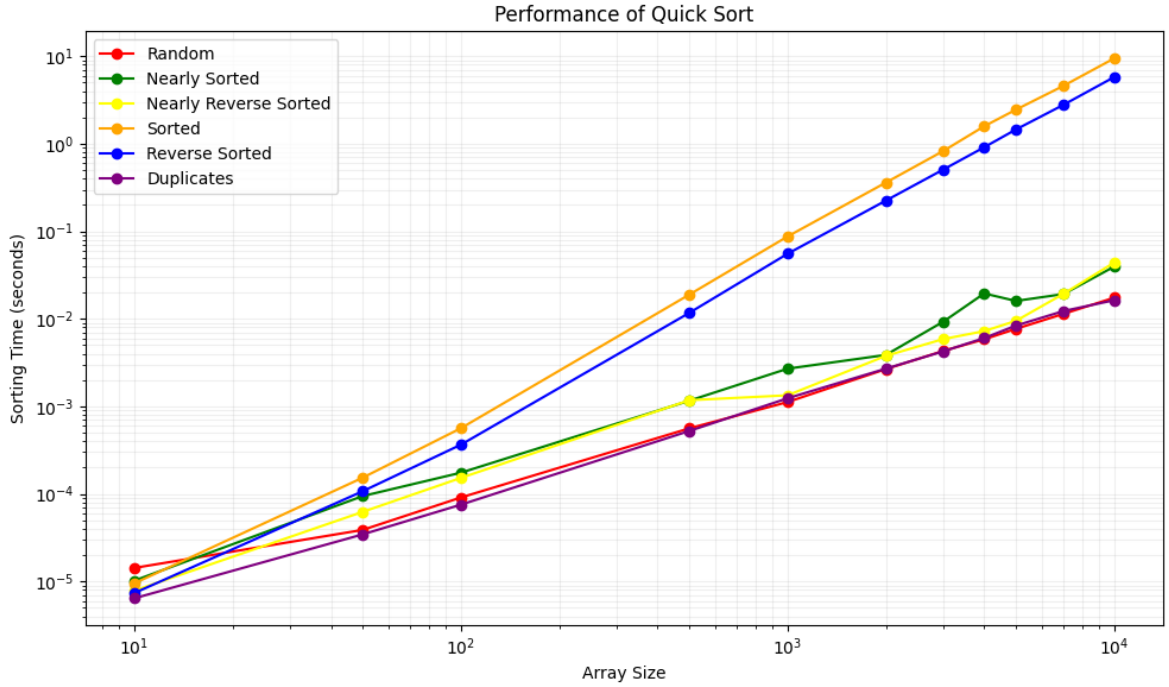

Figure 1: Performance of Quick Sort across different input types

Based on the measurement results, Quick Sort demonstrates excellent performance for random arrays and arrays with duplicates, consistently being among the fastest algorithms for these input types. However, its performance degrades significantly for sorted and reverse-sorted arrays, where it exhibits quadratic time complexity. For instance, at array size 10,000, Quick Sort took approximately 9.47 seconds for sorted arrays compared to just 0.017 seconds for random arrays. This confirms the theoretical weakness of Quick Sort when dealing with already sorted or nearly sorted data.

2.2 Merge Sort

Merge Sort is another divide-and-conquer algorithm that divides the input array into two halves, recursively sorts them, and then merges the sorted halves. The key operation is the merging of two sorted subarrays into a single sorted array. Merge Sort has a consistent O(n log n) time complexity for all cases, making it reliable across different input types.

2.2.1 Implementation

2.2.2 Results

The measurement results show that Merge Sort maintains consistent performance across all input types, with execution times clustering closely together regardless of the array's initial arrangement. This sta-

```
def merge(arr, left, mid, right):
    n1 = mid - left + 1
    n2 = right - mid
    # Create temp arrays
    L = [0] * n1
    R = [0] * n2
    # Copy data to temp arrays L[] and R[]
    for i in range(n1):
        L[i] = arr[left + i]
    for j in range(n2):
        R[j] = arr[mid + 1 + j]
    i = 0  # Initial index of first subarray
    j = 0  # Initial index of second subarray
    k = left  # Initial index of merged subarray
    # Merge the temp arrays back
    # into arr[left..right]
    while i < n1 and j < n2:
        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1
    # Copy the remaining elements of L[],
    # if there are any
    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1
    # Copy the remaining elements of R[],
    # if there are any
    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1
    return


def merge_sort_rec(arr, left, right):
    if left < right:
        mid = (left + right) // 2
        merge_sort_rec(arr, left, mid)
        merge_sort_rec(arr, mid + 1, right)
        merge(arr, left, mid, right)
    return


def merge_sort(arr):
    merge_sort_rec(arr, 0, len(arr)-1)
```

Listing 2: Implementation of Merge Sort

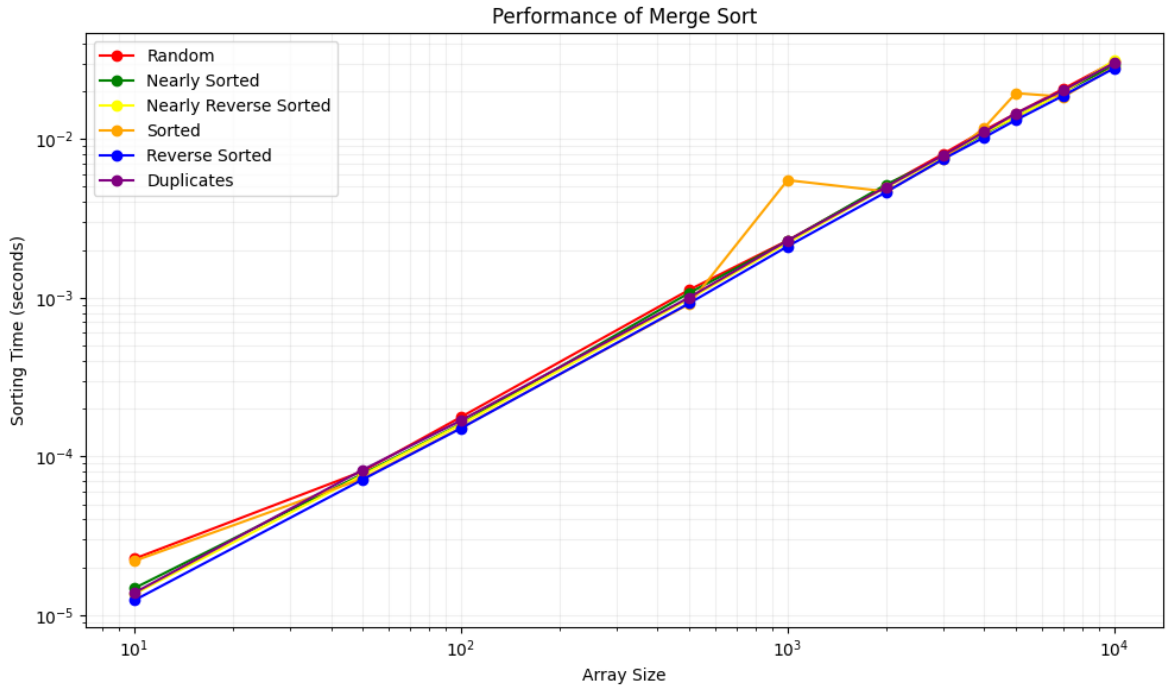

Figure 2: Performance of Merge Sort across different input types

bility makes Merge Sort particularly valuable when predictable performance is required. For example, at array size 10,000, Merge Sort's execution time ranges from 0.028 to 0.031 seconds across different input types, demonstrating its stability. However, it's generally not the fastest algorithm for any particular input type, usually being outperformed by Quick Sort for random arrays and by Smart Sort for sorted arrays.

2.3 Heap Sort

Heap Sort uses a binary heap data structure to sort elements. It first builds a max-heap from the input data, then repeatedly extracts the maximum element from the heap and rebuilds the heap until all elements are extracted. Heap Sort has a consistent O(n log n) time complexity in all cases and requires O(1) extra space.

2.3.1 Implementation

2.3.2 Results

The measurement results indicate that Heap Sort exhibits consistent performance across all input types, similar to Merge Sort. It performs particularly well for nearly reverse-sorted arrays, where it often outperforms other algorithms. For example, at array size 10,000, Heap Sort took approximately 0.031 seconds for nearly reverse-sorted arrays, making it competitive with Merge Sort. However, it's generally

```python
def heapify(arr, n, i):
    largest = i
    l_idx = 2 * i + 1
    r_idx = 2 * i + 2
    if l_idx < n and arr[l_idx] > arr[largest]:
        largest = l_idx
    if r_idx < n and arr[r_idx] > arr[largest]:
        largest = r_idx
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]  # Swap
        heapify(arr, n, largest)
```

```python
def heap_sort(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n - 1, 0, -1):
        arr[0], arr[i] = arr[i], arr[0]
        heapify(arr, i, 0)
    return
```

Listing 3: Implementation of Heap Sort

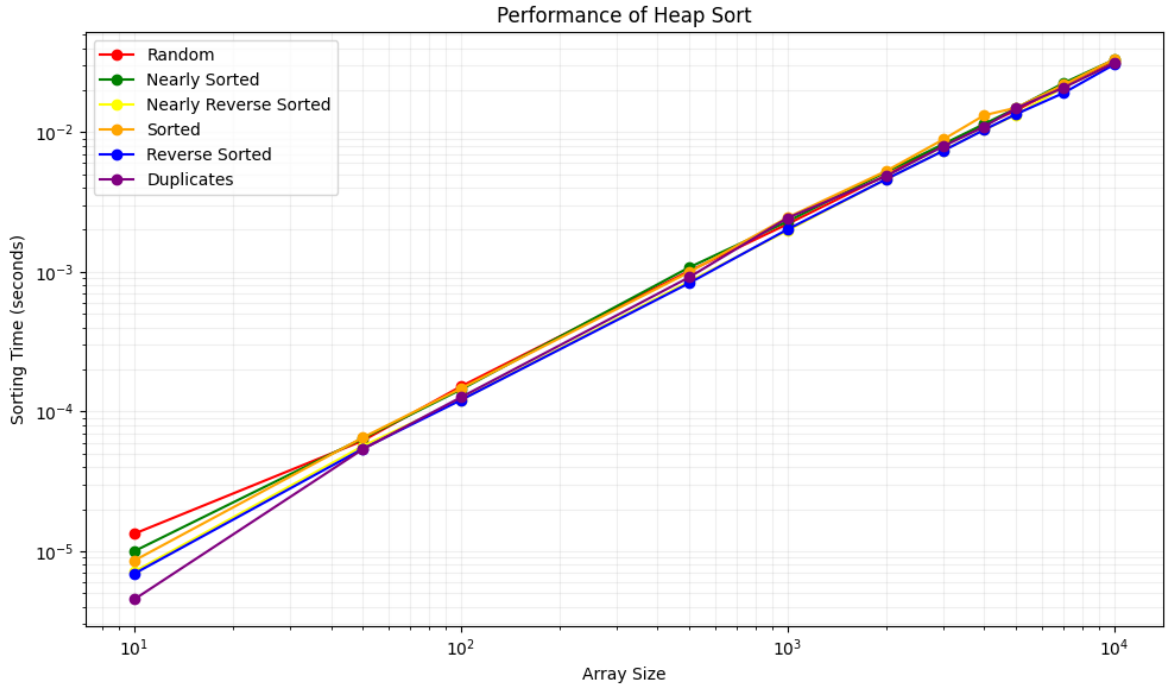

Figure 3: Performance of Heap Sort across different input types

slower than Quick Sort for random arrays and significantly slower than Smart Sort for sorted arrays.

2.4 Patience Sort

Patience Sort is inspired by the card game Patience (Solitaire). It creates piles of cards and uses a merge operation similar to Merge Sort to combine them. The algorithm has a worst-case time complexity of $O(n^2)$ if unoptimized, and O(n log n) if optimized with a heap for merging. It performs particularly well on partially sorted data, and has a best case O(n) complexity when the array is sortred in reverse.

2.4.1 Implementation

```
def patience_sort(collection):
    piles = []  # A list of piles of cards
    for item in collection:
        new_pile = [item]
        for pile in piles:
            if item < pile[-1]:  # If the item is smaller than the last card in
            ↪  the pile
                pile.append(item)  # Add the item to the pile
                break
        else:
            piles.append(new_pile)  # If the item is larger than all the piles,
            ↪  create a new pile for it

    sorted_list = []  # A list of sorted cards
    while piles:
        smallest_pile = min(piles, key=lambda pile: pile[-1])  # Find the
        ↪  smallest pile by comparing the last cards
        sorted_list.append(smallest_pile.pop())  # Remove the last card from the
        ↪  smallest pile and add it to the sorted list
        if not smallest_pile:  # If the smallest pile is empty, remove it from
        ↪  the list of piles
            piles.remove(smallest_pile)

    for i in range(len(collection)):
        collection[i] = sorted_list[i]

    return
```

Listing 4: Implementation of Patience Sort

2.4.2 Results

Patience Sort demonstrates interesting performance characteristics across different input types. It performs reasonably well for random arrays and arrays with duplicates, although not as fast as Quick Sort or Smart Sort. However, it struggles significantly with sorted and nearly sorted arrays, showing exponential growth in execution time as the array size increases. For instance, at array size 10,000, Patience

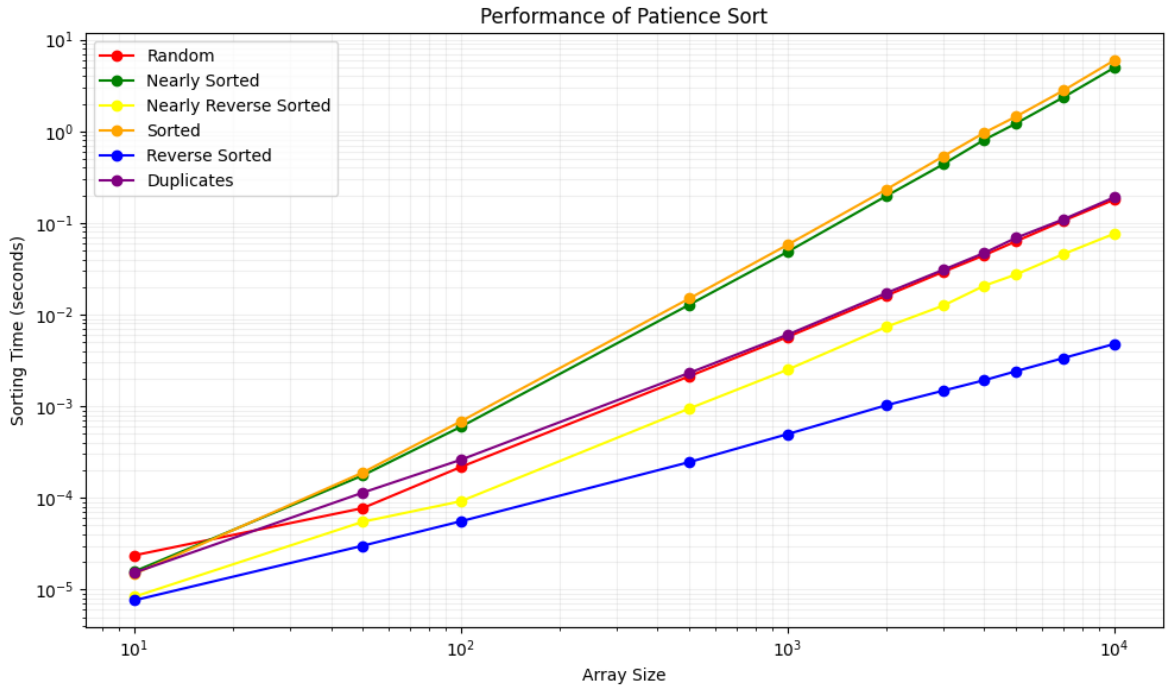

Figure 4: Performance of Patience Sort across different input types

Sort took approximately 5.95 seconds for sorted arrays and 4.95 seconds for nearly sorted arrays, making it the second slowest algorithm for these input types. Interestingly, Patience Sort performs well on reverse-sorted arrays, being the second fastest algorithm after Smart Sort for this input type, confirming the theory around it.

2.5 Smart Sort

Smart Sort is hybrid sorting algorithm that adapts its strategy based on the input characteristics, it demonstrates exceptional performance in certain scenarios, particularly for sorted and reverse-sorted arrays. It uses Heap sort for arrays smaller than 500 elements, and merge for the rest in the case of nearly sorted arrays. For all the other cases it uses QuickSort.

2.5.1 Implementation

2.5.2 Results

The measurement results confirm that Smart Sort is exceptionally efficient for sorted and reverse-sorted arrays, outperforming all other algorithms by several orders of magnitude. For example, at array size 10,000, Smart Sort took just 0.000678 seconds for sorted arrays and 0.001199 seconds for reverse-sorted arrays, while the next fastest algorithm took over 0.027 seconds. This is because Smart Sort includes optimization for detecting already sorted sequences. For random arrays and arrays with duplicates,

```
def smart_sort(arr):
    size = len(arr)

    # Handle empty or single element arrays
    if size <= 1:
        return

    # Check if array is already sorted
    if is_sorted(arr):
        return

    # Check if array is reverse sorted
    if is_sorted(arr[::-1]):
        # Reverse the array in-place
        left, right = 0, len(arr) - 1
        while left < right:
            arr[left], arr[right] = arr[right], arr[left]
            left += 1
            right -= 1
        return

    # For very small arrays (size < 50)
    if size < 50:
        quick_sort(arr)
        return

    nearly_sorted = is_nearly_sorted_v2(arr)

    if nearly_sorted:
        # For arrays size 50-500
        if size <= 500:
            heap_sort(arr)
            return

        # For larger arrays (size >= 500)
        else:
            merge_sort(arr)
            return

    quick_sort(arr)
    return
```

Listing 5: Implementation of Smart Sort

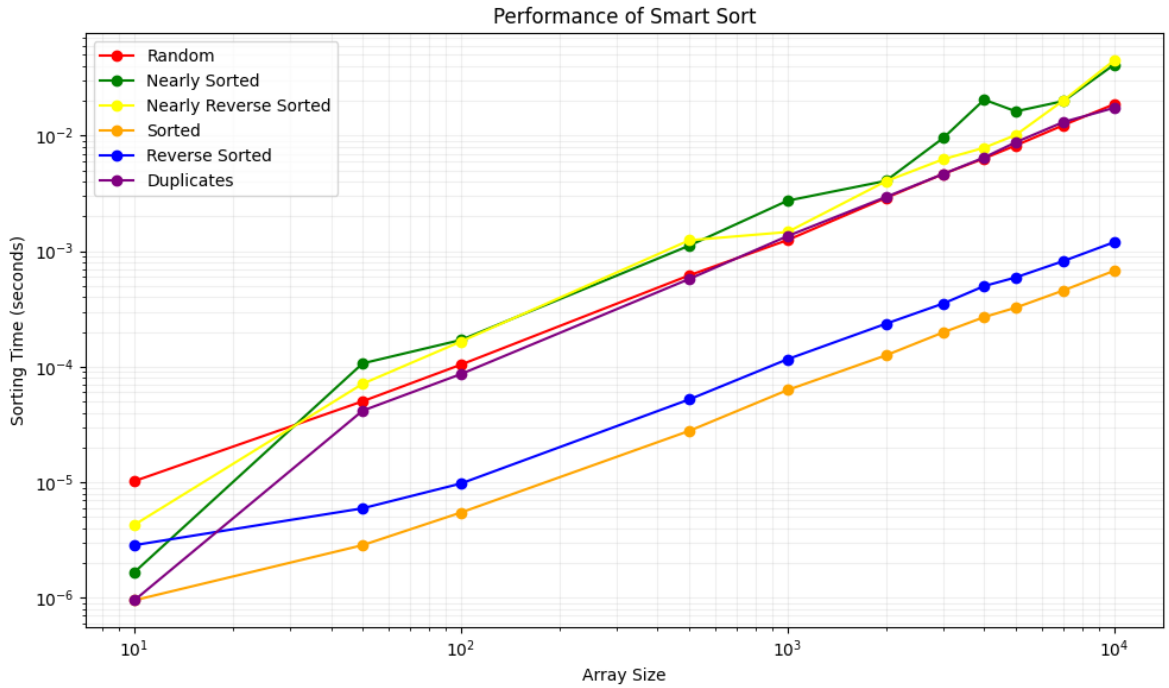

Figure 5: Performance of Smart Sort across different input types

Smart Sort performs similarly to Quick Sort, being slightly slower but still very efficient. However, for nearly sorted arrays, Smart Sort is somewhat slower than Merge Sort and Heap Sort at larger array sizes. Even if the Smart Sort we implemented picks the best sort for the job, the part of analyzing the input slows it down, making it 2nd best performing in most cases.

3 Conclusions

Based on the empirical analysis of the five sorting algorithms across different input types and sizes, we can draw the following conclusions:

1. **Algorithm Selection Depends on Input Characteristics**: No single algorithm performs best across all input types. The choice of sorting algorithm should be informed by the expected characteristics of the input data.

2. **Quick Sort**: Excels with random arrays and arrays with duplicates, making it a good general-purpose choice when the input is expected to be unsorted. However, it performs poorly on sorted or nearly sorted arrays, exhibiting quadratic time complexity in these cases.

3. **Merge Sort**: Provides consistent performance across all input types, making it reliable when predictable execution time is required. While not the fastest in most scenarios, its stability makes it valuable for applications where worst-case performance is a concern.

4. **Heap Sort**: Similar to Merge Sort in offering consistent performance across input types. It performs particularly well for nearly reverse-sorted arrays but is generally outperformed by other algorithms in specific scenarios.

5. **Patience Sort**: Demonstrates good performance for random arrays and reverse-sorted arrays but struggles significantly with sorted and nearly sorted arrays. Its use should be limited to scenarios where the input is known to be random or reverse-sorted.

6. **Smart Sort**: Shows exceptional performance for sorted and reverse-sorted arrays, suggesting sophisticated detection of already sorted sequences. It performs comparably to Quick Sort for random arrays and arrays with duplicates, making it a versatile choice when the input characteristics are known.

7. **Scaling Behavior**: All algorithms show the expected O(n log n) scaling for random arrays, as evidenced by the linear relationship on the log-log plots. However, Quick Sort and Patience Sort exhibit quadratic scaling for sorted arrays, confirming their theoretical worst-case behavior.

8. **Practical Takeaways**:

    - For general-purpose sorting with unknown input characteristics, Merge Sort provides the most consistent performance.
    - When sorting efficiency is critical and the input is likely to be random, Quick Sort is the best choice.
    - even if the SmartSort we implemented picks the best sort for the job, the part of analyzing the input slows it down, making it 2nd best performing in most cases.
    - For memory-constrained environments, Heap Sort provides good performance with minimal additional space requirements.

The results imply the importance of understanding both the theoretical properties of sorting algorithms and their empirical performance characteristics when selecting an algorithm for a specific application.