Ministerul Educației și Cercetării al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Laboratory Work 2

**Study and empirical analysis of sorting algorithms.
Analysis of quickSort, mergeSort, heapSort, GnomeSort**

Elaborated:
st. gr. FAF-233 Chirtoaca Liviu

Verified:
asist. univ. Fiștic Cristofor

Chișinău - 2025

Contents

1 Introduction

This document provides a detailed analysis of four sorting algorithms: QuickSort, MergeSort, HeapSort, and Gnome Sort. Each algorithm is explained concisely, and their performance is compared using time complexity and key features. A summary table and comprehensive conclusion are included to guide the selection of the most appropriate algorithm for specific use cases.

2 Algorithm Explanations

2.1 QuickSort

- **Type**: Divide-and-conquer
- **How it works**:
  - Chooses a pivot element.

```
int pivot = Partition(arr, left, right);
QuickSort(arr, left, pivot - 1);
QuickSort(arr, pivot + 1, right);
```

Figure 1: Pivot selection

  - Partitions the array into two halves: elements less than the pivot and elements greater than the pivot.

```
int pivot = arr[right];
int i = left - 1;
for (int j = left; j < right; j++)
    if (arr[j] <= pivot) Swap(arr, ++i, j);
Swap(arr, i + 1, right);
return i + 1;
```

Figure 2: Partitions

  - Recursively sorts the two halves.
- **Time Complexity**:
  - Best/Average: $O(n \log n)$
  - Worst: $O(n^2)$ (if pivot selection is poor, e.g., already sorted data)
- **Key Feature**: Fastest for random data, in-place, cache-friendly.

2.2 MergeSort

- **Type**: Divide-and-conquer
- **How it works**:
  - Splits the array into two halves.
  - Recursively sorts each half.
  - Merges the two sorted halves into a single sorted array.
- **Time Complexity**:
  - Best/Average/Worst: $O(n \log n)$
- **Key Feature**: Stable, consistent performance, requires extra memory.

2.3 HeapSort

- **Type**: Comparison-based
- **How it works**:
  - Builds a max-heap from the array.

```
int largest = i, left = 2 * i + 1, right = 2 * i + 2;
if (left < n && arr[left] > arr[largest]) largest = left;
if (right < n && arr[right] > arr[largest]) largest = right;
if (largest != i)
```

Figure 3: Heapify

  - Repeatedly extracts the maximum element and places it at the end of the array.

```
int n = arr.Length;
for (int i = n / 2 - 1; i >= 0; i--)
    Heapify(arr, n, i);
for (int i = n - 1; i > 0; i--)
{
    Swap(arr, 0, i);
    Heapify(arr, i, 0);
}
```

Figure 4: Sort

  - Maintains the heap property after each extraction.
- **Time Complexity**:
  - Best/Average/Worst: $O(n \log n)$
- **Key Feature**: In-place, not stable, good for memory-constrained systems.

2.4 Gnome Sort

- **Type**: Exotic algorithm
- **How it works**:
  - Scans the array from left to right.
  - If the current element is smaller than the previous, swaps them and moves backward.
  - Otherwise, moves forward.

```
int pos = 0;
while (pos < arr.Length)
{
    if (pos == 0 || arr[pos] >= arr[pos - 1])
        pos++;
    else
        Swap(arr, pos, --pos);
```

Figure 5: GnomeSort

- **Time Complexity**:
  - Best: $O(n)$ (already sorted)
  - Average/Worst: $O(n^2)$
- **Key Feature**: Simple, adaptive (fast for nearly-sorted data), inefficient for large datasets.

3 Summary Table

Algorithm	Best Case	Average Case	Worst Case	Key Feature
QuickSort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	Fastest for random data
MergeSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Stable, consistent
HeapSort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	In-place, memory-efficient
Gnome Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Simple, adaptive, slow for large data

Table 1: Comparison of Sorting Algorithms

4 Results

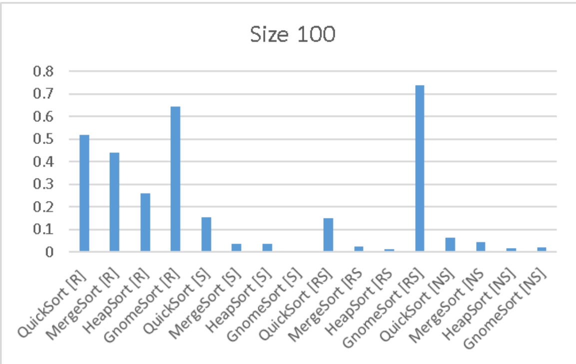

Figure 6: 100 Numbers

QuickSort [R]	Random	100	0.5171
MergeSort [R]	Random	100	0.4399
HeapSort [R]	Random	100	0.2583
GnomeSort [F	Random	100	0.644
QuickSort [S]	Sorted	100	0.1536
MergeSort [S]	Sorted	100	0.0352
HeapSort [S]	Sorted	100	0.0358
GnomeSort [S	Sorted	100	0.0026
QuickSort [RS	ReverseSorted	100	0.1508
MergeSort [R!	ReverseSorted	100	0.0271
HeapSort [RS	ReverseSorted	100	0.0151
GnomeSort [F	ReverseSorted	100	0.7362
QuickSort [NS	NearlySorted	100	0.064
MergeSort [N	NearlySorted	100	0.0444
HeapSort [NS	NearlySorted	100	0.0165
GnomeSort [N	NearlySorted	100	0.0202

Figure 7: CSV Data

Description: This chart shows the performance of sorting algorithms on small datasets (100 elements).

Behavior:

- **Random (R):** Algorithms like HeapSort and MergeSort perform well, but Gnome Sort may struggle due to its $O(n^2)$ complexity.
- **Sorted (S):** Gnome Sort excels here with $O(n)$ complexity, while QuickSort may degrade to $O(n^2)$ if the pivot selection is poor.
- **Reverse Sorted (Rs):** Similar to sorted data, Gnome Sort, MergeSort and HeapSort performs well, but QuickSort may struggle.
- **Nearly Sorted (Ns):** Gnome Sort performs well due to its adaptive nature, while HeapSort and MergeSort show consistent performance.

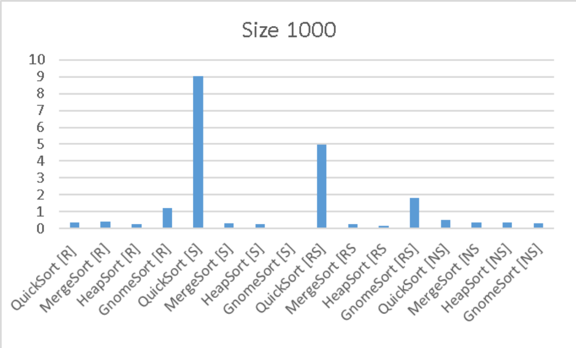

Figure 8: 1000 Numbers

QuickSort [R]	Random	1000	0.3605
MergeSort [R]	Random	1000	0.3991
HeapSort [R]	Random	1000	0.2505
GnomeSort [F	Random	1000	1.2316
QuickSort [S]	Sorted	1000	9.0237
MergeSort [S]	Sorted	1000	0.3119
HeapSort [S]	Sorted	1000	0.2423
GnomeSort [S	Sorted	1000	0.0681
QuickSort [RS	ReverseSorted	1000	4.9649
MergeSort [R	ReverseSorted	1000	0.2535
HeapSort [RS	ReverseSorted	1000	0.1732
GnomeSort [F	ReverseSorted	1000	1.8396
QuickSort [NS	NearlySorted	1000	0.5289
MergeSort [N	NearlySorted	1000	0.357
HeapSort [NS	NearlySorted	1000	0.3463
GnomeSort [N	NearlySorted	1000	0.336

Figure 9: CSV Data

Description: This chart shows the performance of sorting algorithms on medium-sized datasets (1000 elements).

Behavior:

- **Random (R):** QuickSort, HeapSort and MergeSort dominate due to their $O(n \log n)$ complexity. Gnome Sort starts to show its inefficiency.
- **Sorted (S):** QuickSort performs poorly due to its pivot taking, but HeapSort and MergeSort are faster due to their efficient divide-and-conquer approach.
- **Reverse Sorted (Rs):** Similar to sorted data, but QuickSort may degrade further if the pivot selection is poor.
- **Nearly Sorted (Ns):** Gnome Sort performs better than QuickSort and MergeSort.

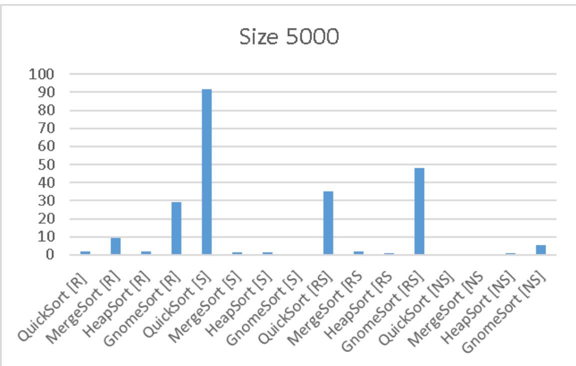

Figure 10: 5000 Numbers

QuickSort [R]	Random	5000	1.7926
MergeSort [R]	Random	5000	9.1516
HeapSort [R]	Random	5000	1.976
GnomeSort [F	Random	5000	29.025
QuickSort [S]	Sorted	5000	91.8547
MergeSort [S]	Sorted	5000	1.212
HeapSort [S]	Sorted	5000	1.6724
GnomeSort [S	Sorted	5000	0.0272
QuickSort [RS	ReverseSorted	5000	34.9085
MergeSort [R	ReverseSorted	5000	1.8869
HeapSort [RS	ReverseSorted	5000	0.9704
GnomeSort [F	ReverseSorted	5000	47.9686
QuickSort [NS	NearlySorted	5000	0.3586
MergeSort [N	NearlySorted	5000	0.4046
HeapSort [NS	NearlySorted	5000	1.1507
GnomeSort [N	NearlySorted	5000	5.274

Figure 11: CSV Data

Description: This chart shows the performance of sorting algorithms on larger datasets (5000 elements).

Behavior:

- **Random (R):** QuickSort and HeapSort continue to dominate, while Gnome Sort becomes impractical due to its $O(n^2)$ complexity.
- **Sorted (S):** Gnome Sort is the fastest due to its best case being O(n) which is more efficient than HeapSort and MergeSort.
- **Reverse Sorted (Rs):** QuickSort improved from the random sort but still is significantly worse than MergeSort and HeapSort.
- **Nearly Sorted (Ns):** Gnome Sort performs better than on random data but is still much slower than QuickSort and MergeSort.

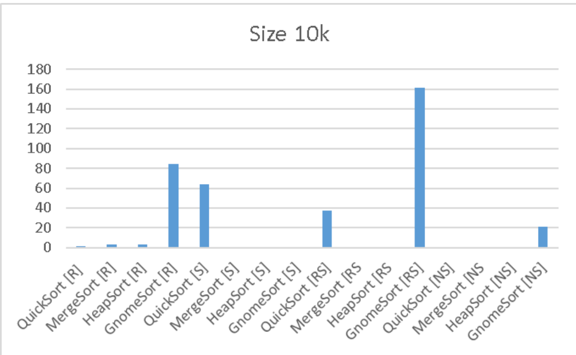

Figure 12: 10,000 Numbers

QuickSort [R]	Random	10000	1.0594
MergeSort [R]	Random	10000	3.0434
HeapSort [R]	Random	10000	3.2849
GnomeSort [F	Random	10000	84.2178
QuickSort [S]	Sorted	10000	63.6705
MergeSort [S]	Sorted	10000	0.296
HeapSort [S]	Sorted	10000	0.6729
GnomeSort [S	Sorted	10000	0.0327
QuickSort [RS	ReverseSorted	10000	37.5478
MergeSort [R	ReverseSorted	10000	0.3584
HeapSort [RS	ReverseSorted	10000	0.8257
GnomeSort [F	ReverseSorted	10000	161.2952
QuickSort [NS	NearlySorted	10000	0.5103
MergeSort [N	NearlySorted	10000	0.6459
HeapSort [NS	NearlySorted	10000	0.8441
GnomeSort [N	NearlySorted	10000	20.8377

Figure 13: CSV Data

Description: This chart shows the performance of sorting algorithms on very large datasets (10,000 elements).

Behavior:

- **Random (R):** QuickSort and MergeSort are the clear winners, with Gnome Sort being extremely slow.
- **Sorted (S):** Gnome Sort is efficient but impractical for large datasets yet for sorted arrays will always perform better then this 3.
- **Reverse Sorted (Rs):** QuickSort may degrade to $O(n^2)$, while MergeSort remains consistent, Gnome Sort showing its weaknesses.
- **Nearly Sorted (Ns):** Gnome Sort is slightly better than on random data but still far slower than this 3.

5 Conclusion

The analysis of sorting algorithms reveals distinct strengths and weaknesses for each approach, highlighting the importance of selecting the right algorithm based on the specific use case and data characteristics.

- **QuickSort** is the fastest for general-purpose sorting, especially on random data, due to its efficient partitioning and in-place nature. However, its performance degrades to $O(n^2)$ in the worst case, particularly with poor pivot selection on already sorted or nearly sorted data.
- **MergeSort** provides consistent $O(n \log n)$ performance across all cases, making it ideal for scenarios requiring stable sorting or predictable performance. Its main drawback is the additional memory requirement for merging.
- **HeapSort** is an excellent choice for memory-constrained environments, as it operates in-place with guaranteed $O(n \log n)$ performance. However, it is not stable and has higher constant factors compared to QuickSort and MergeSort.
- **Gnome Sort**, while simple and adaptive, is inefficient for large datasets due to its $O(n^2)$ average and worst-case complexity. It performs well on small or nearly sorted data, making it suitable for niche applications or educational purposes.

In practice, the choice of sorting algorithm depends on the specific requirements:

- Use **QuickSort** for general-purpose sorting of large, random datasets.
- Use **MergeSort** when stability or consistent performance is critical.
- Use **HeapSort** in memory-constrained systems or for priority queue implementations.
- Use **Gnome Sort** only for small or nearly sorted datasets, or as a teaching tool to demonstrate adaptive sorting.

This analysis underscores the trade-offs between simplicity, performance, and memory usage, emphasizing that no single algorithm is universally optimal. Understanding the characteristics of the input data and the requirements of the application is key to selecting the most appropriate sorting algorithm.