Ministry of Education of Republic of Moldova
Technical University of Moldova
Faculty of Computers, Informatics and Microelectronics

Study and empirical analysis of sorting algorithms.

Verified:
Fistic Cristofor asist. univ.

Amza Vladislavi

Moldova, March 2025

Contents

1 Algorithm Analysis

1.1 Objective

Analysis of quickSort, mergeSort, heapSort, (one of your choice)

1.2 Tasks

- Implement the algorithms listed above in a programming language;
- Establish the properties of the input data against which the analysis is performed;
- Decide the comparison metric for the algorithms;
- Perform empirical analysis of the proposed algorithms;
- Make a graphical presentation of the data obtained;
- Deduce conclusions of the laboratory;

1.3 Theoretical Notes:

Sorting algorithms are a cornerstone of computer science, providing systematic methods to arrange data in a specific order, such as ascending or descending. They are essential for optimizing search operations, organizing data, and solving problems that rely on ordered datasets. Sorting algorithms can be analyzed based on their time complexity, space complexity, stability, and adaptability. Understanding these algorithms is crucial for designing efficient systems and applications.

1.4 Key Concepts in Sorting Algorithms

1.4.1 Time Complexity

This measures the number of operations an algorithm performs relative to the input size (n). Algorithms like Bubble Sort and Insertion Sort have $O(n^2)$ time complexity, making them inefficient for large datasets. In contrast, Merge Sort and Quick Sort achieve $O(n \log n)$ time complexity, making them more suitable for larger inputs. Time complexity is a critical factor when dealing with scalability and performance in real-world applications.

1.4.2 Space Complexity

This refers to the amount of memory an algorithm requires. In-place algorithms like Quick Sort use minimal extra memory, while others like Merge Sort require additional space proportional to the input size. Space complexity is particularly important in environments with limited memory resources, such as embedded systems or mobile devices.

1.4.3 Stability

A sorting algorithm is stable if it preserves the relative order of equal elements. For example, Merge Sort is stable, while Quick Sort is not. Stability is crucial in applications where the initial order of equal elements matters, such as sorting records by multiple keys. A stable algorithm ensures that the original sequence is maintained for equivalent items.

1.4.4 Adaptability

Some algorithms, like Insertion Sort, perform better on partially sorted data, adapting to the input's existing order. Adaptability is a desirable property in scenarios where data is already partially organized, as it reduces the number of operations required to achieve the final sorted order.

1.5 Types of Sorting Algorithms

Sorting algorithms can be broadly classified into two categories:

1.5.1 Comparison-Based Sorting

These algorithms compare elements to determine their order. Examples include Bubble Sort, Merge Sort, and Quick Sort. They are versatile but have a lower bound of $O(n \log n)$ for time complexity. Comparison-based algorithms are widely used due to their flexibility and applicability to various data types.

1.5.2 Non-Comparison-Based Sorting

These algorithms use properties of the data, such as counting or digit manipulation, to sort. Examples include Counting Sort, Radix Sort, and Bucket Sort. They can achieve linear time complexity ($O(n)$) but are limited to specific data types, such as integers or strings. Non-comparison-based algorithms are highly efficient for specialized tasks but lack the generality of comparison-based methods.

1.5.3 Applications and Trade-offs

The choice of sorting algorithm depends on the problem context. For small datasets, simpler algorithms like Insertion Sort may suffice, while for large datasets, more efficient algorithms like Merge Sort or Quick Sort are preferred. Non-comparison-based algorithms excel in specialized scenarios, such as sorting integers within a known range. Understanding the trade-offs between time complexity, space complexity, and stability is essential for selecting the right algorithm for a given problem.

1.6 Why Sorting Algorithms are Important

The sorting algorithm is important in Computer Science because it reduces the complexity of a problem. There is a wide range of applications for these algorithms, including searching algorithms, database algorithms, divide and conquer methods, and data structure algorithms.

In the following sections, we list some important scientific applications where sorting algorithms are used.

- When you have hundreds of datasets you want to print, you might want to arrange them in some way.
- Once we get the data sorted, we can get the k-th smallest and k-th largest item in $(O(1))$ time.
- Searching any element in a huge data set becomes easy. We can use Binary search method for search if we have sorted data. So, Sorting become important here.
- They can be used in software and in conceptual problems to solve more advanced problems.

2 Implementation

All four algorithms will be implemented in their native form in Python and analyzed empirically based on the time required for their completion. While the general trend of the results may be similar to other experimental observations, the particular efficiency in rapport with input will vary depending on the memory of the device used. The error margin determined will constitute 2.5 seconds as per experimental measurement.

2.1 Quick Sort

Quick Sort is a divide-and-conquer algorithm that picks an element as a pivot and partitions the array around the pivot. This partitioning step is recursive, resulting in two subarrays, which are sorted independently.

2.1.1 Python Implementation

```python
def lomuto_partition(arr, low, high):
    pivot = arr[high]
    i = low
    for j in range(low, high):
        if arr[j] < pivot:
            arr[i], arr[j] = arr[j], arr[i]
            i += 1
    arr[i], arr[high] = arr[high], arr[i]
    return i

def quick_sort(arr, low=0, high=None):
    if high is None:
        high = len(arr) - 1
    if low < high:
        pivot_index = lomuto_partition(arr, low, high)
        quick_sort(arr, low, pivot_index - 1)
        quick_sort(arr, pivot_index + 1, high)
    return arr
```

Listing 2.1: *Quick Sort*

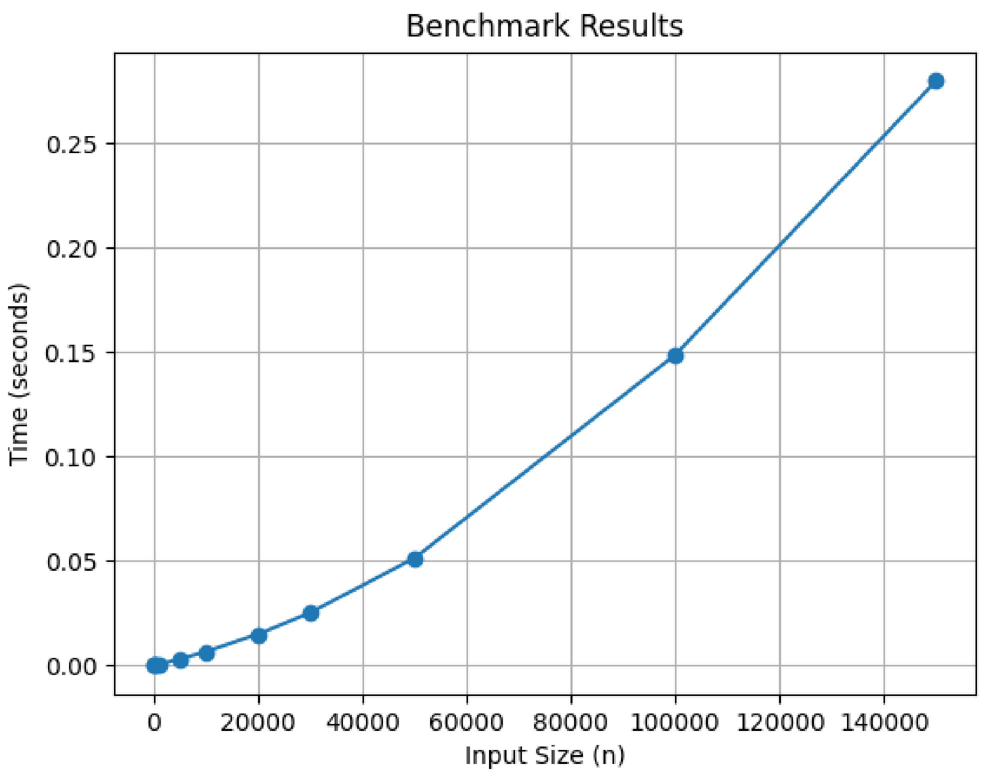

Figure 2.1: *Quick Sort graph*

Time Complexity: $O(n \log n)$
Auxiliary Space: $O(\log n)$

2.2 Merge Sort

Merge Sort is another divide-and-conquer algorithm. It divides the array into two halves, recursively sorts each half, and then merges them back together in a sorted manner.

2.2.1 Python Implementation

```
def merge(left, right):
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result.extend(left[i:])
    result.extend(right[j:])
    return result

def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left_half = merge_sort(arr[:mid])
    right_half = merge_sort(arr[mid:])

    return merge(left_half, right_half)
```

Listing 2.2: *Merge Sort*

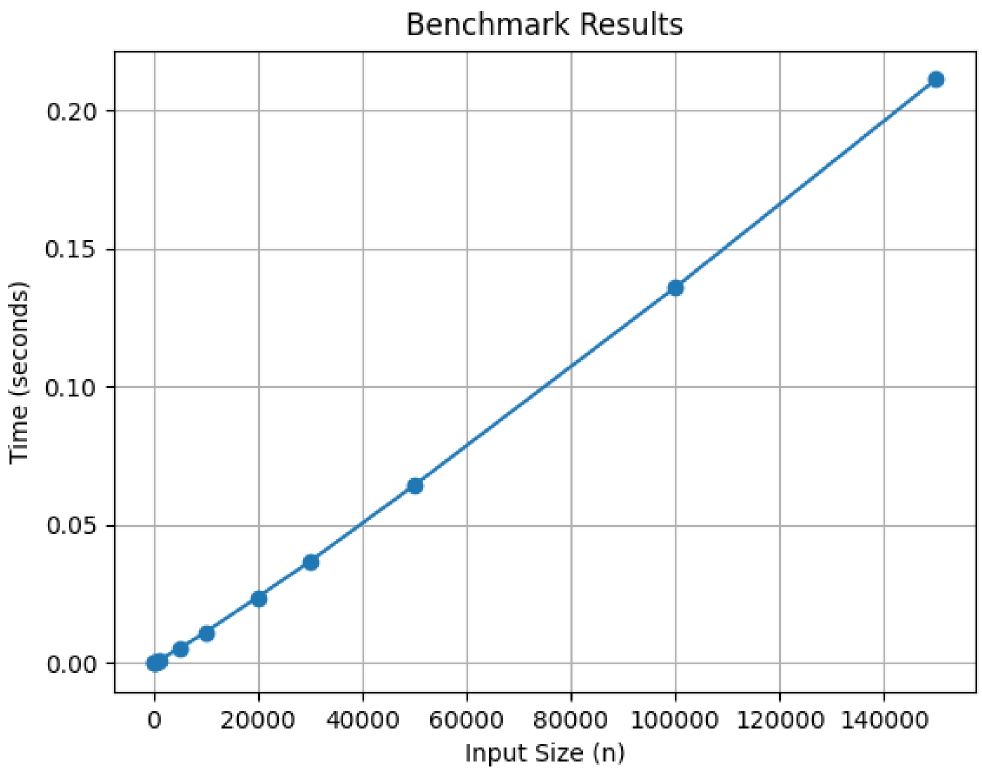

Figure 2.2: *Merge Sort graph*

Time Complexity: $O(n \log n)$
Auxiliary Space: $O(n)$

2.3 Heap Sort

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure. It first builds a max-heap and repeatedly extracts the maximum element to place it in the sorted array.

2.3.1 Python Implementation

```python
def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    if left < n and arr[largest] < arr[left]:
        largest = left

    if right < n and arr[largest] < arr[right]:
        largest = right

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]

        heapify(arr, n, largest)

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[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
```

Listing 2.3: *Heap Sort*

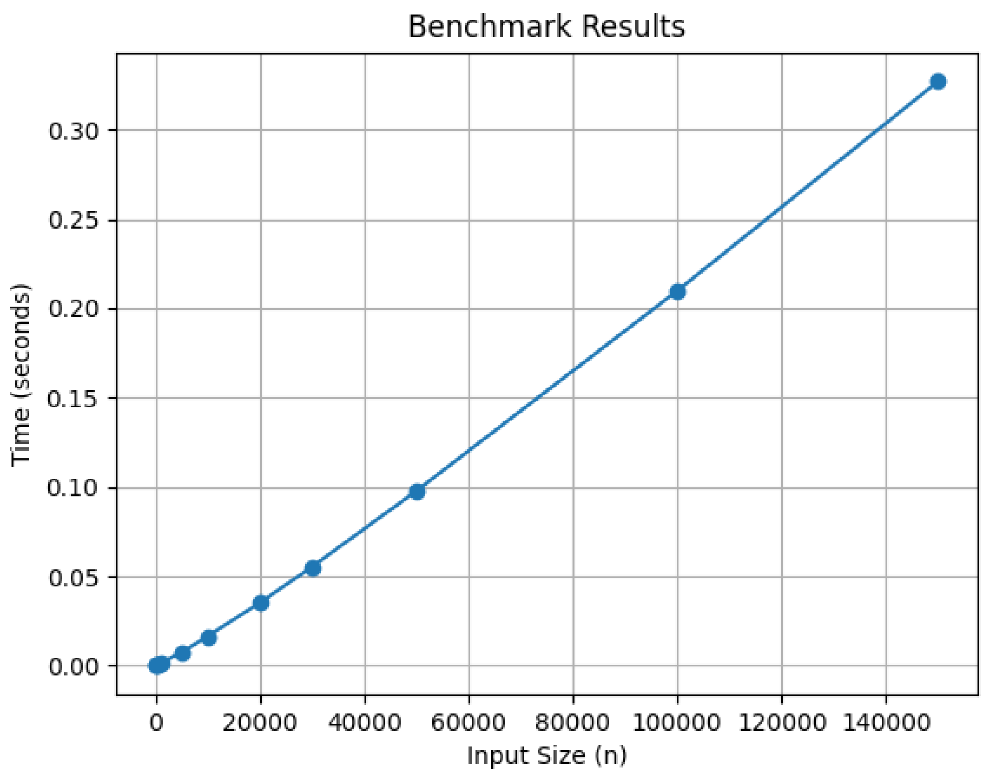

Figure 2.3: *Heap Sort graph*

Time Complexity: $O(n \log n)$
Auxiliary Space: $O(1)$

2.4 Bitonic Sort

Bitonic Sort is a parallel sorting algorithm based on the bitonic sequence. It works by recursively creating a bitonic sequence from the input array and then sorting the sequence using bitonic merges.

2.4.1 Python Implementation

```python
def bitonic_compare(arr, low, cnt, direction):
    half = cnt // 2
    for i in range(low, low + half):
        if (arr[i] > arr[i + half]) == direction:
            arr[i], arr[i + half] = arr[i + half], arr[i]

def bitonic_merge(arr, low, cnt, direction):
    if cnt > 1:
        bitonic_compare(arr, low, cnt, direction)
        half = cnt // 2
        bitonic_merge(arr, low, half, direction)
        bitonic_merge(arr, low + half, half, direction)
```

```python
def bitonic_sort(arr, low, cnt, direction):
    if cnt > 1:
        half = cnt // 2
        bitonic_sort(arr, low, half, True)
        bitonic_sort(arr, low + half, half, False)
        bitonic_merge(arr, low, cnt, direction)

def bitonic_sort_check(arr):
    n = len(arr)
    if n & (n - 1) != 0:
        raise ValueError("Bitonic sort requires array length to be a
            ↪ power of 2.")

    bitonic_sort(arr, 0, n, True)
    return arr
```

Listing 2.4: *Bitonic Sort*

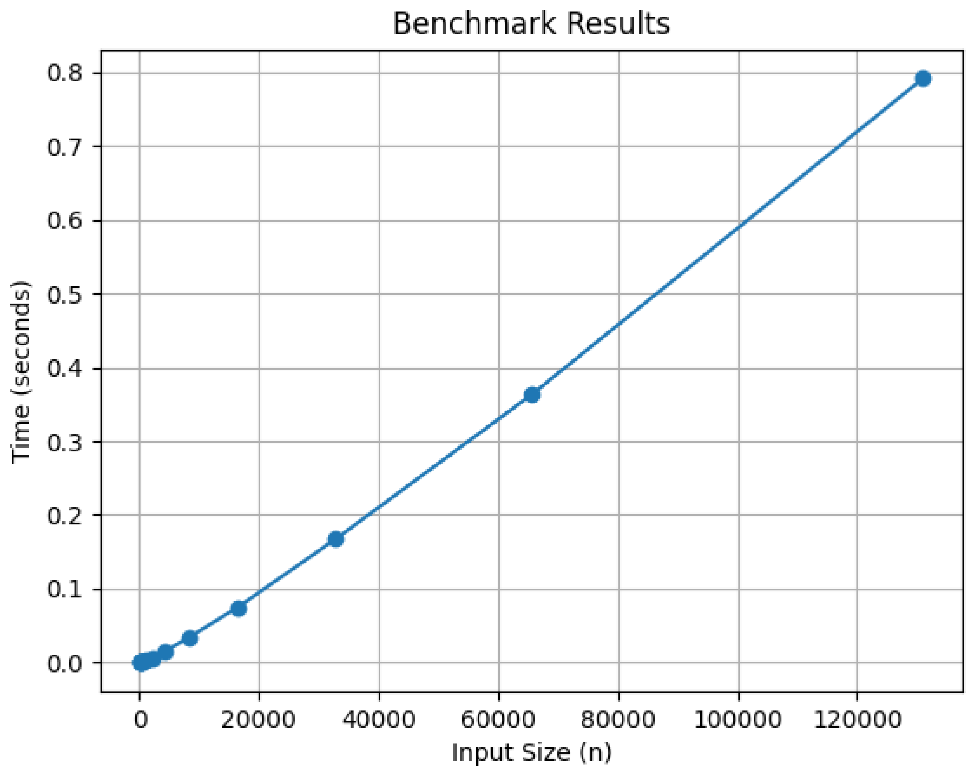

Figure 2.4: *Bitonic Sort graph*

Time Complexity: $O(\log^2 n)$
Auxiliary Space: $O(n)$

3 Conclusion

Sorting algorithms are fundamental tools in computer science, essential for organizing data to facilitate efficient searching, data processing, and optimization. The choice of sorting algorithm has a direct impact on performance, especially with larger datasets. Here, we have examined four important sorting algorithms: Quick Sort, Heap Sort, Merge Sort, and Bitonic Sort. Each of these algorithms has distinct characteristics, advantages, and disadvantages that make them suited to specific scenarios.

3.1 Quick Sort

Quick Sort is a divide-and-conquer algorithm that efficiently sorts an array by selecting a pivot element and partitioning the array around this pivot. Its average time complexity of $O(n \log n)$ and low overhead for practical applications make it one of the fastest sorting algorithms in practice. However, Quick Sort suffers from a worst-case time complexity of $O(n^2)$, which can occur when the pivot selection is poor, such as when the array is already sorted. Despite this, its efficiency and simplicity, along with optimizations like random pivot selection or median-of-three pivoting, make Quick Sort highly effective for general-purpose sorting tasks.

3.2 Heap Sort

Heap Sort utilizes a binary heap data structure to sort an array. It works by first building a heap and then repeatedly extracting the maximum (or minimum) element from the heap, resulting in a sorted array. With a time complexity of $O(n \log n)$ in both the average and worst cases, Heap Sort guarantees stable performance regardless of the input order. However, it typically has more overhead than Quick Sort due to the need to build and maintain the heap structure. Additionally, it is an in-place sorting algorithm, but its performance is often slower than Quick Sort in practice due to constant factors. Nonetheless, Heap Sort is particularly useful when a guaranteed worst-case performance is required or in systems with limited memory.

3.3 Merge Sort

Merge Sort is another divide-and-conquer algorithm that divides an array into smaller subarrays, recursively sorts them, and then merges the sorted subarrays. Its time complexity of $O(n \log n)$ is consistent across best, average, and worst cases, making it a reliable and stable sorting algorithm. Merge Sort is particularly advantageous for large datasets and external sorting, where data resides outside the main memory. However, Merge Sort requires additional space for the temporary subarrays, making it less efficient in terms of memory usage compared to in-place sorting algorithms like Quick Sort and Heap Sort. Despite its memory overhead, its guaranteed performance and stability make it a great choice for sorting linked lists and in environments where consistent time complexity is crucial.

3.4 Bitonic Sort

Bitonic Sort is a comparison-based sorting algorithm that is particularly efficient on parallel computing architectures. It works by recursively combining and sorting two-bitonic sequences, which are sequences that first increase and then decrease. With a time complexity of $O(n \log^2 n)$, Bitonic Sort is less efficient than Quick Sort, Merge Sort, and Heap Sort for general-purpose sorting tasks. However, its parallelizable nature makes it ideal for hardware implementations, such as on Graphics Processing Units (GPUs) or specialized parallel hardware. The algorithm's non-comparative nature in certain implementations and its ability to perform sorting in parallel make it useful for highly parallel systems, though it is rarely used in practice for general-purpose sorting due to its higher time complexity and inefficiency in sequential applications.

3.5 Comparative Analysis

While all these algorithms achieve $O(n \log n)$ time complexity under various conditions, the differences in their practical performance arise from factors like constant factors, memory usage, stability, and parallelizability.

- **Quick Sort** is generally the fastest in practice for average use cases, especially with optimizations like random pivot selection and tail recursion elimination. However, it is not stable and can degrade in performance for certain input sequences.
- **Heap Sort** offers the advantage of guaranteed $O(n \log n)$ performance in the worst case, but it is often slower than Quick Sort in practice due to its constant overhead and lack of locality of reference.
- **Merge Sort** is preferred when stability is important or for sorting linked lists and large datasets that cannot fit entirely into memory. Its guaranteed time complexity and stability make it highly reliable, although it is more memory-intensive.

- **Bitonic Sort**, while not suitable for general-purpose sorting due to its $O(n \log^2 n)$ time complexity, is invaluable for parallel computing environments. Its ability to leverage parallel processing makes it an interesting choice for specialized applications that require massive parallelism.

3.6 Conclusion

In summary, the selection of the appropriate sorting algorithm depends largely on the specific problem at hand, including factors like the dataset size, available memory, the need for parallelization, and the importance of algorithm stability. For most practical, general-purpose sorting tasks, Quick Sort and Merge Sort offer excellent performance, with Quick Sort favored for smaller datasets and Merge Sort for applications requiring consistent performance. Heap Sort offers reliable worst-case performance, and Bitonic Sort shines in specialized, parallel processing environments. Understanding the strengths and weaknesses of each algorithm allows for better decision-making and optimization when implementing sorting in software systems.