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.

Belih Dmitrii

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 Theory

2.1 Memory usage patterns and index sorting

When the size of the array to be sorted approaches or exceeds the available primary memory, so that (much slower) disk or swap space must be employed, the memory usage pattern of a sorting algorithm becomes important, and an algorithm that might have been fairly efficient when the array fit easily in RAM may become impractical. In this scenario, the total number of comparisons becomes (relatively) less important, and the number of times sections of memory must be copied or swapped to and from the disk can dominate the performance characteristics of an algorithm. Thus, the number of passes and the localization of comparisons can be more important than the raw number of comparisons, since comparisons of nearby elements to one another happen at system bus speed (or, with caching, even at CPU speed), which, compared to disk speed, is virtually instantaneous.

For example, the popular recursive quicksort algorithm provides quite reasonable performance with adequate RAM, but due to the recursive way that it copies portions of the array it becomes much less practical when the array does not fit in RAM, because it may cause a number of slow copy or move operations to and from disk. In that scenario, another algorithm may be preferable even if it requires more total comparisons.

One way to work around this problem, which works well when complex records (such as in a relational database) are being sorted by a relatively small key field, is to create an index into the array and then sort the index, rather than the entire array. (A sorted version of the entire array can then be produced with one pass, reading from the index, but often even that is unnecessary, as having the sorted index is adequate.) Because the index is much smaller than the entire array, it may fit easily in memory where the entire array would not, effectively eliminating the disk-swapping problem. This procedure is sometimes called ”tag sort”.

Another technique for overcoming the memory-size problem is using external sorting, for example, one of the ways is to combine two algorithms in a way that takes

advantage of the strength of each to improve overall performance. For instance, the array might be subdivided into chunks of a size that will fit in RAM, the contents of each chunk sorted using an efficient algorithm (such as quicksort), and the results merged using a k-way merge similar to that used in merge sort. This is faster than performing either merge sort or quicksort over the entire list.

Techniques can also be combined. For sorting very large sets of data that vastly exceed system memory, even the index may need to be sorted using an algorithm or combination of algorithms designed to perform reasonably with virtual memory, i.e., to reduce the amount of swapping required.

2.2 Sorting Basics

- **In-place Sorting:** An in-place sorting algorithm uses constant space for producing the output (modifies the given array only. *Examples: Selection Sort, Bubble Sort, Insertion Sort and Heap Sort.*
- Internal Sorting: Internal Sorting is when all the data is placed in the main memory or internal memory. In internal sorting, the problem cannot take input beyond allocated memory size.
- **External Sorting:** External Sorting is when all the data that needs to be sorted need not to be placed in memory at a time, the sorting is called external sorting. External Sorting is used for the massive amount of data. *For example Merge* sort can be used in external sorting as the whole array does not have to be present all the time in memory,
- **Stable sorting:** When two same items appear in the same order in sorted data as in the original array called stable sort. *Examples: Merge Sort, Insertion Sort, Bubble Sort.*
- **Hybrid Sorting:** A sorting algorithm is called Hybrid if it uses more than one standard sorting algorithms to sort the array. The idea is to take advantages of multiple sorting algorithms. *For example IntroSort uses Insertions sort and Quick Sort.*

2.3 Quick Sort

QuickSort is a sorting algorithm based on the Divide and Conquer that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

2.3.1 How does QuickSort Algorithm work?

QuickSort works on the principle of divide and conquer, breaking down the problem into smaller sub-problems.
There are mainly three steps in the algorithm:

- Choose a Pivot: Select an element from the array as the pivot. The choice of pivot can vary (e.g., first element, last element, random element, or median).
- Partition the Array: Rearrange the array around the pivot. After partitioning, all elements smaller than the pivot will be on its left, and all elements greater than the pivot will be on its right. The pivot is then in its correct position, and we obtain the index of the pivot.
- Recursively Call: Recursively apply the same process to the two partitioned sub-arrays (left and right of the pivot).
- Base Case: The recursion stops when there is only one element left in the sub-array, as a single element is already sorted.

Here's a basic overview of how the QuickSort algorithm works.

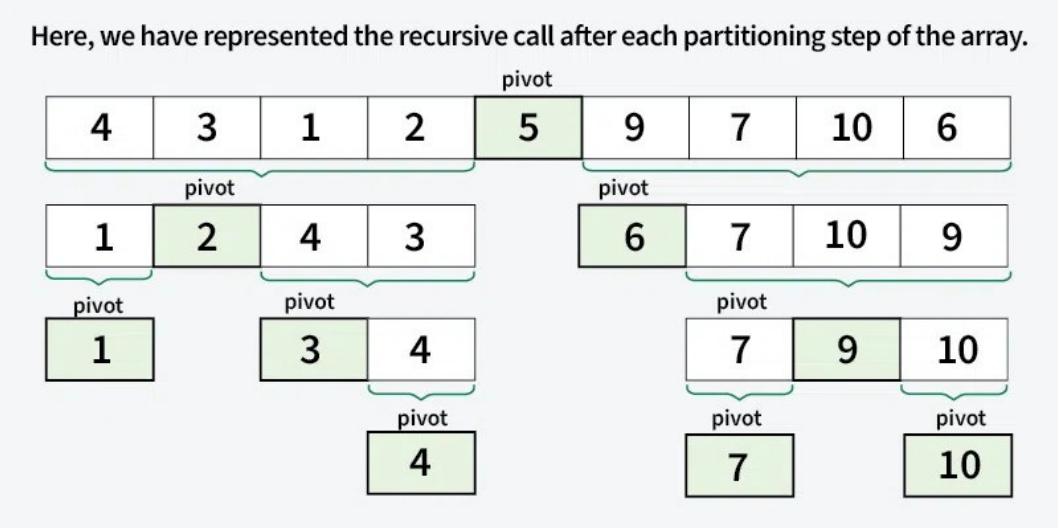

Figure 2.1: *QuickSort algorithm*

2.3.2 Choice of Pivot

There are many different choices for picking pivots in Quick Sort. The choice of pivot significantly impacts the algorithm's performance.

2.3.3 Common Pivot Selection Strategies

- **First or Last Element as Pivot**: The simplest approach is to always pick the first or last element as the pivot. However, this strategy leads to the worst-case time complexity of $O(n^2)$ when the array is already sorted or nearly sorted.
- **Random Element as Pivot**: Picking a random element as the pivot is a preferred approach because it avoids patterns that lead to the worst-case scenario. This randomization improves the average-case performance.
- **Median Element as Pivot**: Selecting the median element as the pivot is ideal in terms of time complexity, as it ensures the array is divided into two equal halves.

However, finding the median requires additional computation, which increases the average time due to higher constants.

2.3.4 Partition Algorithm

The key process in Quick Sort is the partitioning step. There are several algorithms to partition the array, all of which have $O(n)$ time complexity.

2.3.5 Common Partition Algorithms

- **Naive Partition**: This approach creates a copy of the array, places all smaller elements first, followed by all larger elements, and then copies the temporary array back to the original array. It requires $O(n)$ extra space.
- **Lomuto Partition**: This is a simple and widely used partitioning algorithm. It keeps track of the index of smaller elements and performs swaps to partition the array. It is easy to implement but less efficient than Hoare's partition.
- **Hoare's Partition**: This is the fastest partitioning algorithm. It traverses the array from both ends, swapping elements to ensure smaller elements are on the left and larger elements are on the right. It is more efficient than Lomuto's partition but slightly more complex to implement.

2.3.6 Complexity Analysis of Quick Sort

Quick Sort's performance varies depending on the input and pivot selection.

2.3.7 Time Complexity

- **Best Case**: $\Omega(n \log n)$, occurs when the pivot divides the array into two equal halves.
- **Average Case**: $\Theta(n \log n)$, occurs on average when the pivot divides the array into two parts, though not necessarily equal.
- **Worst Case**: $O(n^2)$, occurs when the smallest or largest element is always chosen as the pivot (e.g., in already sorted arrays).

2.3.8 Space Complexity

The auxiliary space required by Quick Sort is $O(n)$ due to the recursive call stack.

2.3.9 Advantages of Quick Sort

- It is a divide-and-conquer algorithm, making it easier to solve problems efficiently.
- It is highly efficient for large datasets.
- It has low overhead, requiring only a small amount of memory.

- It is cache-friendly, as it operates on the same array without copying data to auxiliary arrays.
- It is the fastest general-purpose sorting algorithm for large datasets when stability is not required.
- It is tail-recursive, allowing for tail call optimization.

2.3.10 Disadvantages of Quick Sort

- It has a worst-case time complexity of $O(n^2)$, which occurs when the pivot is chosen poorly.
- It is not suitable for small datasets, as simpler algorithms like Insertion Sort may perform better.
- It is not a stable sort, meaning the relative order of equal elements may not be preserved.

2.3.11 Applications of Quick Sort

- Efficient for sorting large datasets with $O(n \log n)$ average-case time complexity.
- Used in partitioning problems, such as finding the k^{th} smallest element or dividing arrays by a pivot.
- Integral to randomized algorithms, offering better performance than deterministic approaches.
- Applied in cryptography for generating random permutations and unpredictable encryption keys.
- The partitioning step can be parallelized for improved performance in multi-core or distributed systems.
- Important in theoretical computer science for analyzing average-case complexity and developing new techniques.

2.4 Merge Sort

Merge sort is a sorting algorithm that follows the divide-and-conquer approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array.

In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

2.4.1 How does Merge Sort work?

Merge sort is a popular sorting algorithm known for its efficiency and stability. It follows the divide-and-conquer approach to sort a given array of elements. Here's a step-by-step explanation of how merge sort works:

1. Divide: Divide the list or array recursively into two halves until it can no more be divided.
2. Conquer: Each subarray is sorted individually using the merge sort algorithm.
3. Merge: The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.

Illustration of Merge Sort:

```
Lets look at the working of above example:

Divide:

    [38, 27, 43, 10]  is divided into  [38, 27 ] and  [43, 10]  .
    [38, 27]  is divided into  [38]  and  [27]  .
    [43, 10]  is divided into  [43]  and  [10]  .

Conquer:

    [38]  is already sorted.
    [27]  is already sorted.
    [43]  is already sorted.
    [10]  is already sorted.

Merge:

    Merge  [38]  and  [27]  to get  [27, 38]  .
    Merge  [43]  and  [10]  to get  [10,43]  .
    Merge  [27, 38]  and  [10,43]  to get the final sorted list  [10, 27, 38, 43]

Therefore, the sorted list is  [10, 27, 38, 43]  .
```

Listing 2.1: *Merge Sort*

2.4.2 Recurrence Relation of Merge Sort

The recurrence relation of Merge Sort describes the time complexity of the algorithm in terms of its input size n. It is given by:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{if } n > 1 \end{cases}$$

2.4.3 Explanation of the Recurrence Relation

- $T(n)$ represents the total time taken by the algorithm to sort an array of size n.
- $2T\left(\frac{n}{2}\right)$ represents the time taken to recursively sort the two halves of the array. Each half has $\frac{n}{2}$ elements, resulting in two recursive calls.
- $\Theta(n)$ represents the time taken to merge the two sorted halves into a single sorted array.

2.4.4 Complexity Analysis of Merge Sort

Merge Sort has consistent performance across different scenarios, making it a reliable choice for sorting.

2.4.5 Time Complexity

- **Best Case**: $O(n \log n)$, occurs when the array is already sorted or nearly sorted.
- **Average Case**: $O(n \log n)$, occurs when the array is randomly ordered.
- **Worst Case**: $O(n \log n)$, occurs when the array is sorted in reverse order.

2.4.6 Space Complexity

Merge Sort requires $O(n)$ auxiliary space for the temporary array used during the merging process.

2.4.7 Applications of Merge Sort

Merge Sort is widely used in various applications due to its efficiency and stability.

- Sorting large datasets efficiently.
- External sorting, where the dataset is too large to fit in memory.
- Inversion counting, which is useful in statistics and data analysis.
- Used in library methods of programming languages:
  - Its variation, TimSort, is used in Python, Java Android, and Swift. TimSort is preferred for sorting non-primitive types due to its stability, which QuickSort lacks.
  - In Java, `Arrays.sort` uses QuickSort, while `Collections.sort` uses MergeSort.
- Preferred for sorting linked lists due to its divide-and-conquer approach.
- Easily parallelized, as subarrays can be sorted independently and then merged.
- The merge function of Merge Sort is used to solve problems like finding the union and intersection of two sorted arrays.

2.4.8 Advantages and Disadvantages of Merge Sort

Merge Sort has several strengths and weaknesses that make it suitable for specific use cases.

Advantages

- **Stability**: Merge Sort is a stable sorting algorithm, meaning it preserves the relative order of equal elements in the input array.
- **Guaranteed Worst-Case Performance**: Merge Sort has a worst-case time complexity of $O(n \log n)$, making it efficient even for large datasets.

- **Simple to Implement**: The divide-and-conquer approach is straightforward and easy to understand.
- **Naturally Parallel**: The independent sorting of subarrays makes Merge Sort suitable for parallel processing.

Disadvantages

- **Space Complexity**: Merge Sort requires additional memory to store the merged subarrays during the sorting process.
- **Not In-Place**: Merge Sort is not an in-place sorting algorithm, meaning it requires extra memory to store the sorted data. This can be a disadvantage in memory-constrained environments.
- **Slower than QuickSort**: Merge Sort is generally slower than QuickSort because QuickSort is more cache-friendly, as it operates in-place.

2.5 Heap Sort

Heap sort is a comparison-based sorting technique based on Binary Heap Data Structure. It can be seen as an optimization over selection sort where we first find the max (or min) element and swap it with the last (or first). We repeat the same process for the remaining elements. In Heap Sort, we use Binary Heap so that we can quickly find and move the max element in $O(log(n))$ instead of $O(n)$ and hence achieve the $O(nlog(n))$time complexity.

2.5.1 Heap Sort Algorithm

First convert the array into a max heap using heapify, Please note that this happens in-place. The array elements are re-arranged to follow heap properties. Then one by one delete the root node of the Max-heap and replace it with the last node and heapify. Repeat this process while size of heap is greater than 1.

- Rearrange array elements so that they form a Max Heap.
- Repeat the following steps until the heap contains only one element:
    1. Swap the root element of the heap (which is the largest element in current heap) with the last element of the heap.
    2. Remove the last element of the heap (which is now in the correct position). We mainly reduce heap size and do not remove element from the actual array.
    3. Heapify the remaining elements of the heap.
- Finally we get sorted array.

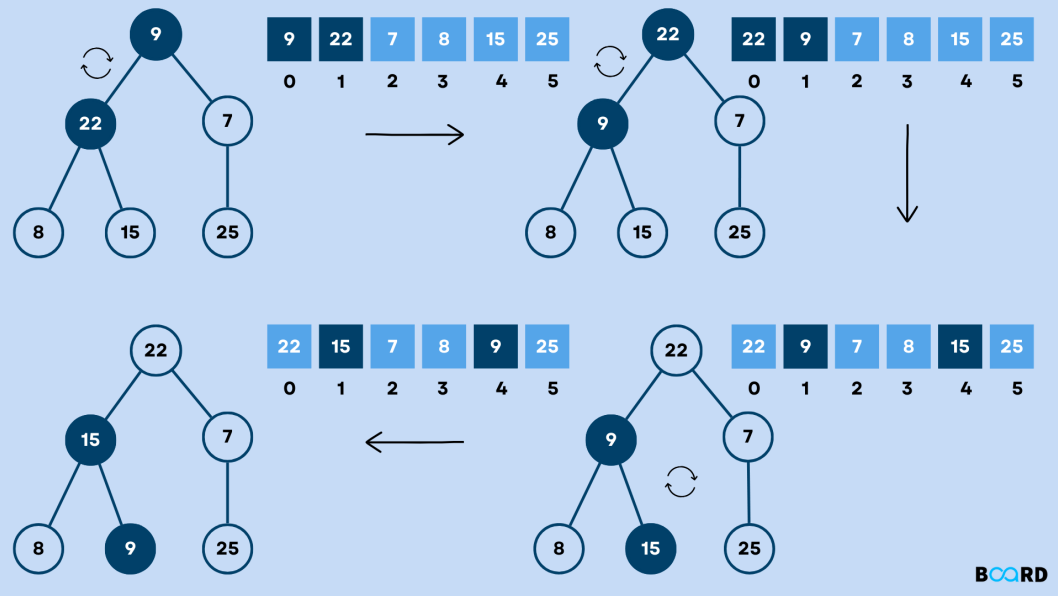

Figure 2.2: *HeapSort algorithm*

2.5.2 Detailed Working of Heap Sort

Step 1: Treat the Array as a Complete Binary Tree
Step 2: Build a Max Heap
Step 3: Sort the array by placing largest element at end of unsorted array.

2.5.3 Complexity Analysis of Heap Sort

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure. Its performance is consistent across different scenarios.

2.5.4 Time Complexity

Heap Sort has a time complexity of $O(n \log n)$ in all cases (best, average, and worst). This is because:

- Building the heap takes $O(n)$ time.
- Extracting the maximum element and heapifying the remaining elements takes $O(\log n)$ time, and this operation is repeated n times.

2.5.5 Space Complexity

- The auxiliary space required by Heap Sort is $O(\log n)$ due to the recursive call stack in the heapify process.
- For an iterative implementation, the auxiliary space can be reduced to $O(1)$, making it an in-place algorithm.

2.5.6 Important Points About Heap Sort

- **In-Place Algorithm**: Heap Sort is an in-place sorting algorithm, meaning it does not require additional memory proportional to the input size.
- **Stability**: The typical implementation of Heap Sort is not stable, but it can be modified to achieve stability.
- **Performance**: Heap Sort is typically 2-3 times slower than well-implemented QuickSort due to its lack of locality of reference, which affects cache performance.

2.5.7 Advantages of Heap Sort

Heap Sort offers several benefits, making it a viable choice for certain applications.

- **Efficient Time Complexity**: Heap Sort has a time complexity of $O(n \log n)$ in all cases, making it efficient for sorting large datasets. The $\log n$ factor comes from the height of the binary heap, ensuring good performance even with a large number of elements.
- **Memory Usage**: Heap Sort can be implemented with minimal memory usage. By using an iterative heapify process instead of a recursive one, the auxiliary space can be reduced to $O(1)$.
- **Simplicity**: Heap Sort is relatively simple to understand compared to other equally efficient sorting algorithms, as it does not rely on advanced concepts like recursion.

2.5.8 Disadvantages of Heap Sort

Despite its advantages, Heap Sort has some limitations that affect its practicality in certain scenarios.

- **Costly**: Heap Sort has higher constants in its time complexity compared to Merge Sort, even though both have the same $O(n \log n)$ time complexity. This makes Heap Sort slower in practice.
- **Unstable**: The typical implementation of Heap Sort is unstable, meaning it may rearrange the relative order of equal elements. While it can be made stable, this requires additional modifications.
- **Inefficient**: Heap Sort is not very efficient due to the high constants in its time complexity, making it slower than QuickSort and Merge Sort in most cases.

2.6 Pigeonhole Sort

Pigeonhole sorting is a sorting algorithm that is suitable for sorting lists of elements where the number of elements and the number of possible key values are approximately the same. It requires $O(n + \text{Range})$ time where n is the number of elements in the input array and Range is the number of possible values in the array.

2.6.1 Working of Algorithm

1. Find the minimum and maximum values in the array. Let the minimum and maximum values be `min` and `max` respectively. Also, find the range as `max−min+1`.
2. Set up an array of initially empty "pigeonholes" the same size as the range.
3. Visit each element of the array and then put each element in its pigeonhole. An element `arr[i]` is put in the hole at index `arr[i] − min`.
4. Start the loop all over the pigeonhole array in order and put the elements from non-empty holes back into the original array.

Pigeonhole sort has limited use as its requirements are rarely met. For arrays where the range is much larger than n, bucket sort is a generalization that is more efficient in space and time.

2.6.2 Complexity Analysis of Pigeonhole Sort

The time complexity of Pigeonhole Sort is $O(n + \text{range})$, where n is the number of elements in the array and range is the range of the input data (i.e., the difference between the maximum and minimum values in the array).
In the given implementation, the algorithm first finds the minimum and maximum values in the array, which takes $O(n)$ time. Then, it calculates the range, which takes constant time. Next, it creates an array of vectors of size equal to the range, which takes constant time. Then, it traverses the input array and puts each element into its respective hole, which takes $O(n)$ time. Finally, it traverses all the holes and puts their elements into the output array in order, which takes $O(\text{range})$ time.
Therefore, the overall time complexity of the algorithm is $O(n + \text{range})$. In the worst case, when the range is significantly larger than the number of elements in the array, the algorithm can be inefficient. However, it can be useful for sorting integer arrays with a relatively small range.
Auxiliary Space: $O(\text{range})$

2.6.3 Advantages of Pigeonhole Sort

- It is a non-comparison-based sort, making it faster in application.
- It is a stable sorting algorithm.
- It performs sorting in linear time.

2.6.4 Disadvantages of Pigeonhole Sort

- It is not easy to know the range of the numbers to sort.
- This algorithm might only work with zero and positive integers.

3 Implementation

3.1 Heap Sort

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

    if l < n and arr[i] < arr[l]:
        largest = l
    if r < n and arr[largest] < arr[r]:
        largest = r

    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)


def heapSort(arr):
    n = len(arr)
    for i in range(n // 2, -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 3.1: *Python implementation Heap Sort*

Explanation: Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure. The 'heapify' function ensures that the subtree rooted at index 'i' is a heap. The 'heapSort' function builds a max-heap and then repeatedly extracts the maximum element from the heap and places it at the end of the array.

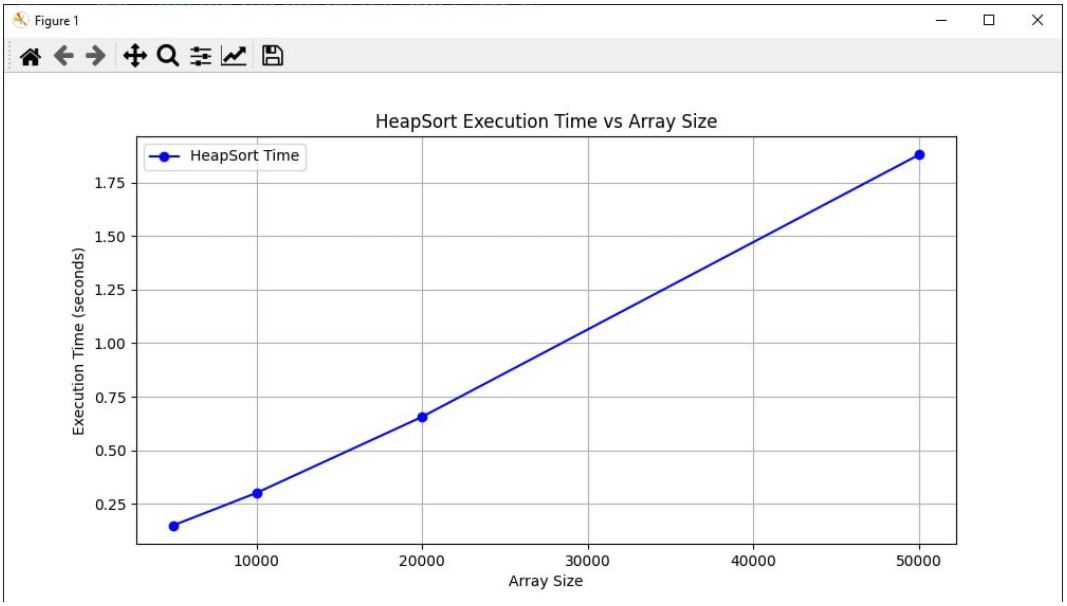

Figure 3.1: *HeapSort Output*

3.2 Quick Sort

3.2.1 Original Quick Sort

```
1  def partition(array, low, high):
2      pivot = array[high]
3      i = low - 1
4      for j in range(low, high):
5          if array[j] <= pivot:
6              i += 1
7              array[i], array[j] = array[j], array[i]
8      array[i + 1], array[high] = array[high], array[i + 1]
9      return i + 1
10
11
12 def quickSort(array, low=0, high=None):
13     if high is None:
14         high = len(array) - 1
15     if low < high:
16         pi = partition(array, low, high)
17         quickSort(array, low, pi - 1)
18         quickSort(array, pi + 1, high)
```

Listing 3.2: *Python implementation Quick Sort*

Explanation: Quick Sort is a divide-and-conquer algorithm. It works by selecting a 'pivot' element from the array and partitioning the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively.

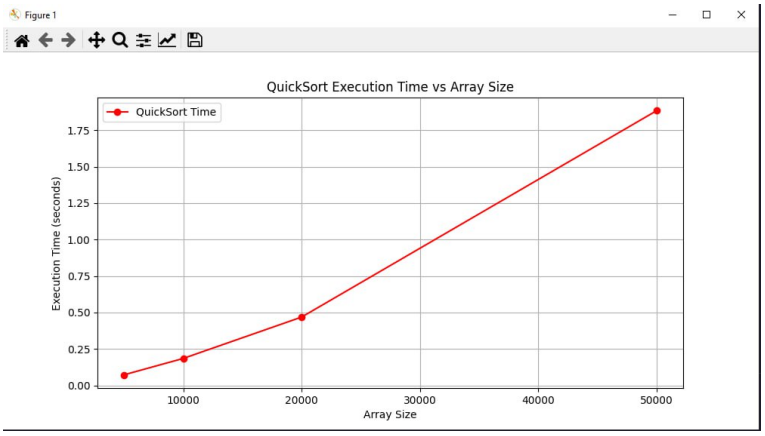

Figure 3.2: *QuickSort Output*

3.2.2 Comprehension Quick Sort

```
1 def quicksort(arr):
2     if len(arr) <= 1:
3         return arr
4     else:
5         pivot = arr[0]
6         left = [x for x in arr[1:] if x < pivot]
7         right = [x for x in arr[1:] if x >= pivot]
8         return quicksort(left) + [pivot] + quicksort(right)
```

Listing 3.3: *Python implementation Comprehension Quick Sort*

Explanation: This version of Quick Sort uses list comprehensions to create the left and right sub-arrays. It is a more Pythonic and concise way of implementing Quick Sort, but it may use more memory due to the creation of new lists.

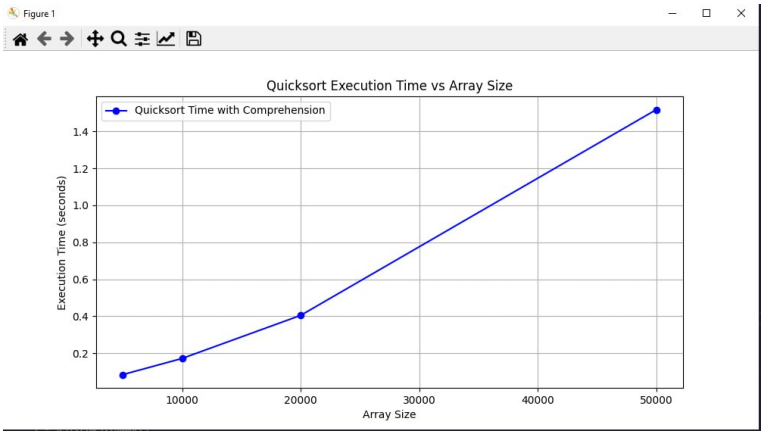

Figure 3.3: *Comprehension QuickSort Output*

3.2.3 Iterative Quick Sort

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


def quicksort(arr):
    stack = [(0, len(arr) - 1)]
    while stack:
        low, high = stack.pop()
        if low < high:
            pi = partition(arr, low, high)
            stack.append((low, pi - 1))
            stack.append((pi + 1, high))
```

Listing 3.4: *Python implementation Iterative Quick Sort*

Explanation: This implementation of Quick Sort uses an explicit stack to simulate recursion. It is useful in environments where recursion depth is limited. The stack keeps track of the sub-arrays that need to be sorted.

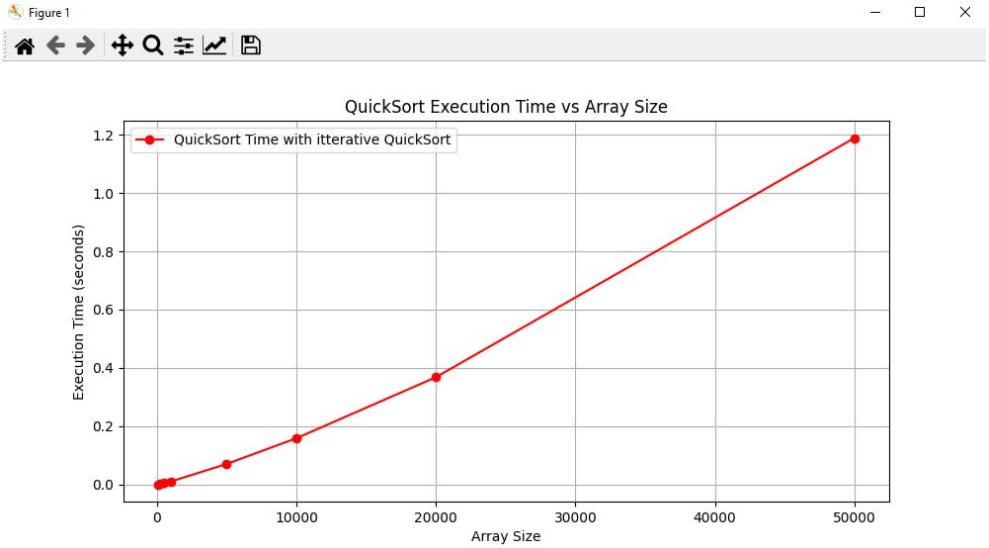

Figure 3.4: *Iterative QuickSort Output*

3.2.4 Inplace Quick Sort

```python
def partition2(arr, low, high):
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] < pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1


def quicksort(arr, low, high):
    if low < high:
        pi = partition2(arr, low, high)
        quicksort(arr, low, pi - 1)
        quicksort(arr, pi + 1, high)
```

Listing 3.5: *Python implementation Inplace Quick Sort*

Explanation: This version of Quick Sort sorts the array in place, meaning it does not require additional memory for storing sub-arrays. It modifies the original array directly, which can be more memory efficient.

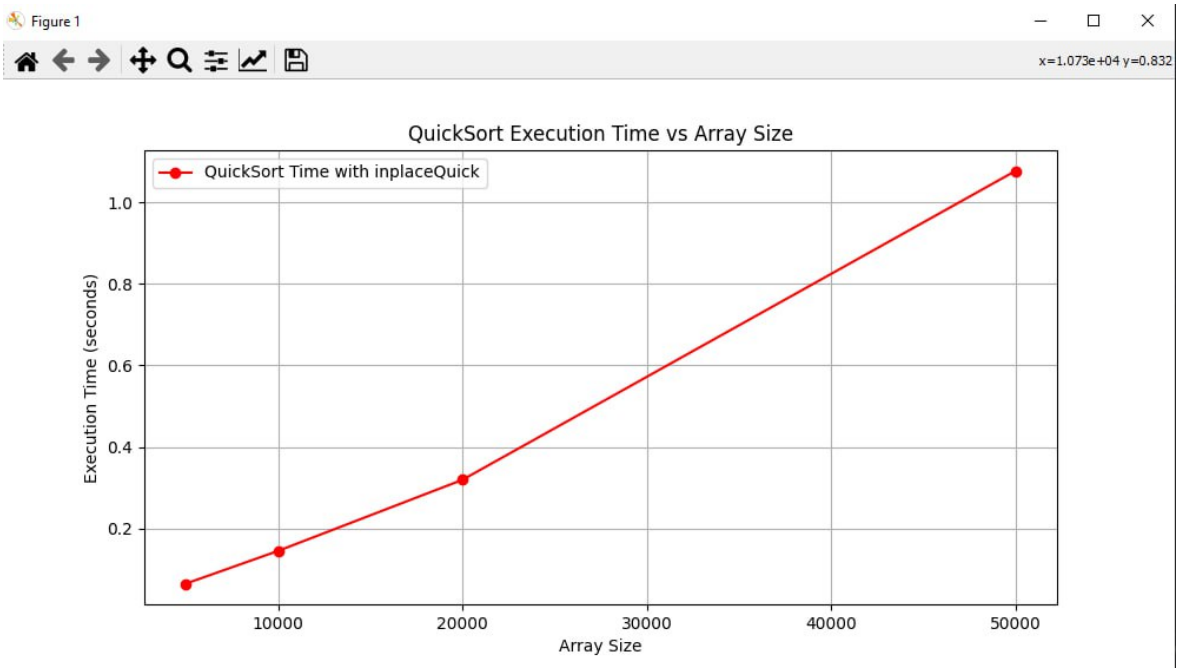

Figure 3.5: *Inplace QuickSort Output*

3.2.5 Hoare Partition Quick Sort

Explanation: Hoare's partition scheme is an alternative to the Lomuto partition scheme.

```python
def partition(arr, low, high):
    pivot = arr[low]
    i = low - 1
    j = high + 1
    while True:
        i += 1
        while arr[i] < pivot:
            i += 1
        j -= 1
        while arr[j] > pivot:
            j -= 1
        if i >= j:
            return j
        arr[i], arr[j] = arr[j], arr[i]


def quicksort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)
        quicksort(arr, low, pi)
        quicksort(arr, pi + 1, high)
```

Listing 3.6: *Python implementation Hoare Partition Quick Sort*

It uses two indices that start at the ends of the array and move towards each other until they meet. This scheme is generally more efficient and results in fewer swaps.

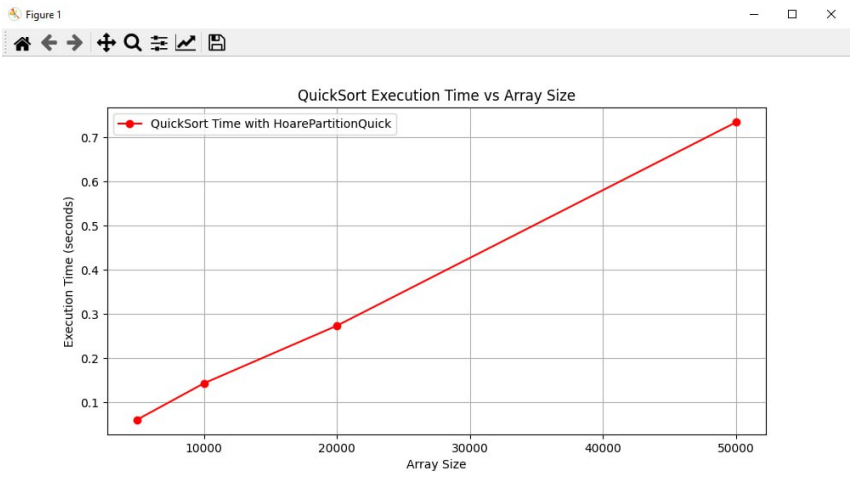

Figure 3.6: *Hoare QuickSort Output*

3.3 Merge Sort

3.3.1 Original Merge Sort

```python
def merge(arr, l, m, r):
    n1 = m - l + 1
    n2 = r - m

    L = [0] * (n1)
    R = [0] * (n2)

    for i in range(0, n1):
        L[i] = arr[l + i]

    for j in range(0, n2):
        R[j] = arr[m + 1 + j]

    # Merge the temp arrays back into arr[l..r]
    i = 0  # Initial index of first subarray
    j = 0  # Initial index of second subarray
    k = l  # Initial index of merged subarray

    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

    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1

    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1

def mergeSort(arr, l, r):
    if l < r:
        # Same as (l+r)//2, but avoids overflow for
        # large l and h
        m = l + (r - l) // 2

        # Sort first and second halves
        mergeSort(arr, l, m)
        mergeSort(arr, m + 1, r)
        merge(arr, l, m, r)
```

Listing 3.7: *Python implementation Merge Sort*

Explanation: Merge Sort is a divide-and-conquer algorithm that divides the input array into two halves, recursively sorts them, and then merges the two sorted halves. The 'merge' function combines two sorted sub-arrays into a single sorted array.

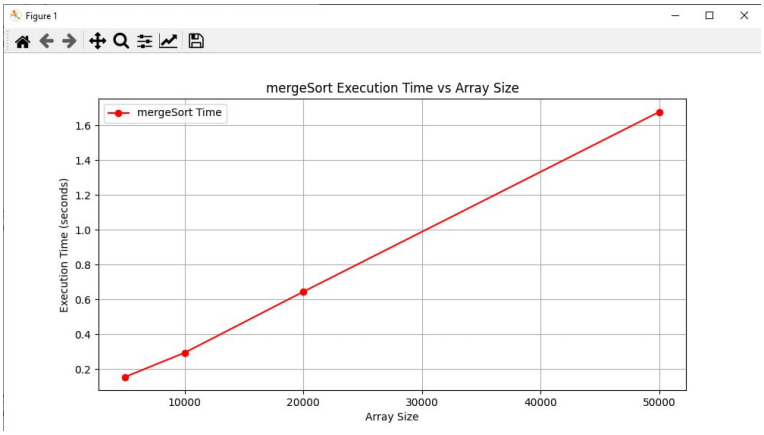

Figure 3.7: *MergeSort Output*

3.3.2 Iterative Merge Sort

```python
def merge(arr1, arr2):
    result = []
    i = j = 0
    while i < len(arr1) and j < len(arr2):
        if arr1[i] < arr2[j]:
            result.append(arr1[i])
            i += 1
        else:
            result.append(arr2[j])
            j += 1
    result.extend(arr1[i:])
    result.extend(arr2[j:])
    return result
def merge_sort_iterative(arr):
    width = 1
    n = len(arr)
    while width < n:
        left = 0
        while left < n:
            mid = min(left + width, n)
            right = min(left + 2 * width, n)
            arr[left:right] = merge(arr[left:mid], arr[mid:right])
            left += 2 * width
        width *= 2
    return arr
```

Listing 3.8: *Python implementation Iterative Merge Sort*

Explanation: The Iterative Merge Sort avoids recursion by using an iterative approach.

It starts by merging small sub-arrays of size 1, then doubles the size in each iteration until the entire array is sorted. The ‘merge‘ function combines two sorted sub-arrays into one. This method is useful in environments where recursion depth is limited and can be more efficient in terms of stack usage.

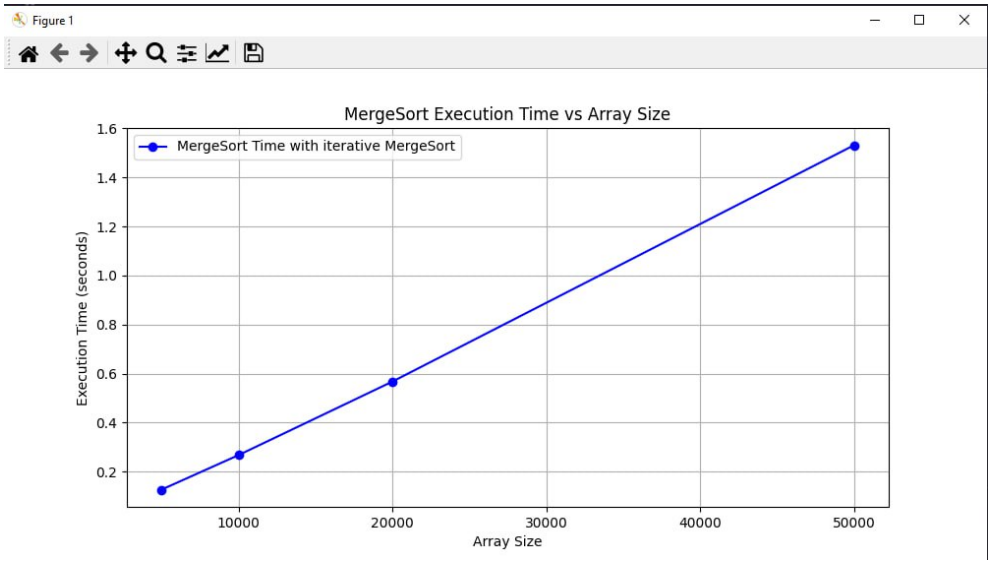

Figure 3.8: *Iterative MergeSort Output*

3.4 Pigeonhole Sort

```
1  def pigeonhole_sort(a):
2      my_min = min(a)
3      my_max = max(a)
4      size = my_max - my_min + 1
5      holes = [0] * size
6      for x in a:
7          assert type(x) is int, "integers only please"
8          holes[x - my_min] += 1
9      i = 0
10     for count in range(size):
11         while holes[count] > 0:
12             holes[count] -= 1
13             a[i] = count + my_min
14             i += 1
```

Listing 3.9: *Python implementation Pigeonhole Sort*

Explanation: Pigeonhole Sort is a sorting algorithm that is suitable for sorting lists of elements where the number of elements and the range of possible key values are approximately the same. It works by determining the range of the values in the input list and creating an array of ”holes” (or buckets) to count the occurrences of each value.

4
Conclusion

In this chapter, we explored various sorting algorithms, each with its unique characteristics, advantages, and disadvantages. Sorting algorithms are fundamental in computer science and are used in a wide range of applications, from database management to data analysis. Below is a comparison of the time and space complexities of the algorithms discussed, followed by a summary of their key features.

4.0.1 Comparison of Sorting Algorithms

Algorithm	Best Case	Average Case	Worst Case	Space Complexity
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$
Pigeonhole Sort	$O(n + \text{range})$	$O(n + \text{range})$	$O(n + \text{range})$	$O(\text{range})$

Table 4.1: *Comparison of Sorting Algorithms by Time and Space Complexity*

4.0.2 Summary of Sorting Algorithms

- **Quick Sort**:
  - Quick Sort is a highly efficient, in-place sorting algorithm with an average time complexity of $O(n \log n)$. However, its worst-case time complexity is $O(n^2)$, which occurs when the pivot selection is poor. Quick Sort is not stable but is widely used due to its cache-friendly nature and low overhead.
- **Merge Sort**:
  - Merge Sort is a stable, divide-and-conquer algorithm with a consistent $O(n \log n)$ time complexity in all cases. It requires $O(n)$ auxiliary space, making it less memory-efficient than Quick Sort. Merge Sort is ideal for external sorting and linked lists due to its stability and parallelizability.
- **Heap Sort**:

  - Heap Sort is an in-place sorting algorithm with a time complexity of $O(n \log n)$ in all cases. It is not stable but is efficient for large datasets. Heap Sort is slower than Quick Sort and Merge Sort in practice due to higher constants in its time complexity.

- **Pigeonhole Sort**:
  - Pigeonhole Sort is a non-comparison-based sorting algorithm with a time complexity of $O(n + \text{range})$. It is efficient for sorting integers with a small range but becomes impractical when the range is large. Pigeonhole Sort is stable and performs sorting in linear time when the range is small.

4.0.3 Final Thoughts

Each sorting algorithm has its strengths and weaknesses, making them suitable for different scenarios. Quick Sort is generally the fastest for large datasets but can suffer from poor pivot selection. Merge Sort is stable and consistent but requires additional memory. Heap Sort is in-place and efficient but slower in practice. Pigeonhole Sort is ideal for small-range integer sorting but is limited in its applicability.
When choosing a sorting algorithm, it is essential to consider the specific requirements of the task, such as the size of the dataset, the range of values, the need for stability, and memory constraints. By understanding the trade-offs between these algorithms, developers can select the most appropriate sorting method for their applications.
https://github.com/DimonBel/Algorithm-Analysis-labs.git