Ministerul Educaţiei 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 Quick Sort

Elaborated: st. gr. FAF-221 Reabciuc Daria-Brianna

Verified: prof. univ. Andrievschi-Bagrin Veronica

asist. univ. Fiștic Cristofor

Chişinău - 2024

Objective

The objective of this work was to implement and analyze the Quick Sort, Merge Sort, Heap Sort and Bubble Sort algorithm in Python, evaluating its performance across datasets of varying sizes through empirical analysis and graphical representation, to understand its efficiency and scalability.

Tasks:

1. Implement the algorithm listed above in a programming language;

2. Establish the properties of the input data against which the analysis is performed; 3. Choose metrics for comparing the algorithm;

4. Perform empirical analysis of the proposed algorithm;

5. Make a graphical representation of the data obtained;

6. Deduce conclusions of the laboratory.

Theoretical Notes:

In theoretical discussions about the analysis of input data for algorithms, particularly sorting algorithms like quick sort, several intrinsic properties of the data are critically examined to predict and understand the algorithm's performance.

The size of the data is a primary consideration, as it directly influences the efficiency of different sorting algorithms. Quick sort, for example, typically operates efficiently on large datasets with its average time complexity of O(n log n). However, the actual performance can vary significantly depending on other factors such as the data's distribution and initial order.

The distribution of the data, whether it be uniform, normal, or skewed, can greatly affect the partitioning process integral to quick sort. A skewed distribution might lead to unbalanced partitions, thereby degrading the algorithm's performance. Similarly, the nature of the data, whether dealing with simple integers or complex objects, impacts the sorting since different types require different levels of comparison complexity.

Another critical factor is the initial ordering of the data. Quick sort can suffer from severely degraded performance, falling to $O(n^2)$ complexity, under certain conditions like when the data is already sorted or nearly sorted, particularly if the pivot selection strategy does not adapt well to these conditions.

This is why the choice of pivot and method of partitioning are pivotal for optimizing quick sort's efficiency. Memory usage also plays a significant role, especially in recursive algorithms like quick sort, which may consume considerable stack space. While quick sort is generally considered in-place, the

depth of recursive calls can vary significantly with the nature and size of the input data.

The concept of stability in sorting, where the original order of equal elements is maintained, is another factor that may be critical depending on the application. Although quick sort is not a stable sort, this may or may not be a significant issue depending on whether there are multiple keys by which the data might be sorted or evaluated.

Additionally, the uniqueness of elements within the dataset can impact the efficiency of sorting. In datasets with many duplicate elements, quick sort's partitioning process may become less efficient, potentially leading to more comparisons and swaps than necessary.

Lastly, access patterns to the data can influence the choice and performance of sorting algorithms. Quick sort, for instance, is more suited to data structures that allow random access, such as arrays, because it frequently accesses disparate parts of the dataset.

Introduction:

In the domain of computer science and data analytics, the efficiency and method of sorting data are crucial for optimizing the functionality of various systems, from simple database management to complex machine learning algorithms. This laboratory work is structured to offer students a thorough understanding and practical experience with four cornerstone sorting algorithms: Quick Sort, Merge Sort, Heap Sort, and Bubble Sort.

Quick Sort is renowned for its efficiency in average-case scenarios and employs a divide-and-conquer strategy, a method conceived by Tony Hoare in 1960. Merge Sort also utilizes a divide-and-conquer approach but stands out for its stability and consistent performance, making it reliable regardless of the dataset's initial condition. Heap Sort deviates from the divide-and-conquer approach, instead leveraging a heap data structure to achieve a uniform O(n log n) performance, distinguishing itself with its unique method of data management. On the other hand, Bubble Sort, known for its straightforward logic, serves as an introductory sorting algorithm, albeit less efficient on larger datasets.

The objective of this lab session is to dissect and understand the distinct characteristics and operational nuances of each sorting technique. Students will explore theoretical concepts such as data distribution effects, initial data ordering, memory consumption, and the principle of algorithmic stability. Practical exercises will allow students to implement each sorting algorithm in Python, apply these implementations across different data sets, and evaluate how variations in data properties impact the performance and efficiency of each algorithm.

This educational endeavor aims to enhance students' comprehension of sorting mechanisms and to cultivate analytical and problem-solving skills relevant to algorithm development and optimization.

Through active participation in these exercises, students will achieve a holistic understanding of the challenges associated with data sorting and the strategic considerations necessary for algorithm selection and application.

Acquiring this knowledge and these skills is imperative for a wide range of applications in computer science and data analytics. As sorting is a fundamental computing challenge, proficiency in these algorithms and an understanding of their specific strengths and weaknesses prepare students for a variety of academic and professional challenges ahead. The lab also emphasizes essential metrics for algorithm comparison, including time complexity, stability, operational efficiency (as measured by the number of comparisons and swaps), and space complexity, providing students with a comprehensive framework for evaluating and choosing appropriate sorting algorithms for different scenarios.

Comparison Metric:

Time Complexity - This measures the algorithm's efficiency in terms of the time it takes to sort an array of a given length. It's crucial for understanding how the algorithm scales with increasing input sizes.

- **Quick Sort**: Has an average and best-case time complexity of O(n log n), but worsens to $O(n^2)$ in the worst-case scenario, particularly when the pivot selection is poor.
- **Merge Sort**: Offers a guaranteed time complexity of O(n log n) in all cases, making it highly predictable and stable in performance across different types of datasets.
- **Heap Sort**: Provides a consistent time complexity of O(n log n), regardless of the initial order of the data, due to the properties of the heap structure.
- **Bubble Sort**: Generally has a poor time complexity of $O(n^2)$, making it inefficient for large datasets, although it is simple to understand and implement.

Space Complexity - This reflects the amount of additional memory the algorithm needs to operate, beyond the input data itself. This metric is vital for evaluating the algorithm's usability in environments with limited memory resources.

- **Quick Sort**: Typically requires O(log n) space because of its recursive stack calls, though this can vary based on the implementation.
- **Merge Sort**: Requires additional memory, typically O(n), because it needs to allocate space for the temporary arrays used during the merge process.
- **Heap Sort**: Operates in-place in the array it sorts, with O(1) additional space, making it memory-efficient.
- **Bubble Sort**: Also operates in-place, requiring only a small, constant amount of additional space (O(1)), primarily for swapping elements.

Stability - This indicates whether the algorithm maintains the relative order of equal elements, which is particularly important when sorting data that have multiple fields or are sorted based on multiple keys sequentially.

- **Quick Sort**: Is not stable; the relative order of equal elements may change during the sorting process.
- **Merge Sort**: Is stable; it maintains the relative order of equal elements, making it suitable for multi-key or complex object sorting.
- **Heap Sort**: Like Quick Sort, it is generally not stable, which might be a concern when sorting records that have multiple comparable fields.
- **Bubble Sort**: Is stable; it naturally maintains the order of equal elements through its swapping mechanism.

These metrics are essential for understanding the trade-offs when choosing among Quick Sort, Merge Sort, Heap Sort, and Bubble Sort for a particular application or dataset.

Input Format:

For a comprehensive comparison and demonstration of how Quick Sort, Merge Sort, Heap Sort, and Bubble Sort operate, we can use input arrays that vary in size and distribution. These examples will illustrate how each algorithm handles small and large datasets, as well as different data distributions:

Small Dataset: A small array can demonstrate how efficiently each algorithm performs on a dataset that is quick to sort due to its size. It can also highlight the overheads associated with more complex algorithms compared to simpler ones.

Example:

small_dataset = random.sample(range(1, 101), 50)

Here, Quick Sort and Merge Sort might show similar efficiency due to their divide-and-conquer approaches, which handle small datasets well. Heap Sort's initial heap construction might be slightly more overhead than necessary for such a small array, while Bubble Sort should perform adequately due to the limited number of elements.

Large Random Dataset: A large array filled with random values tests the algorithms' efficiency and scalability. It reflects average-case scenarios for Quick Sort and Merge Sort, while testing Bubble Sort's limitations due to its $O(n^2)$ complexity.

Example:

large_random_dataset = random.sample(range(1, 10001), 1000) # 1000 unique values between 1 and 10000

In this case, Quick Sort and Merge Sort are expected to perform well, demonstrating their O(n log n) efficiency. Heap Sort should also handle this dataset effectively. However, Bubble Sort is likely to struggle with the dataset's size, highlighting its inefficiency for larger arrays.

Nearly Sorted Dataset: This tests the algorithms' performances on data that is already close to being sorted, which can be particularly challenging for some sorting strategies.

Example:

nearly_sorted_dataset = list(range(1, 1001)) + random.sample(range(1001, 1101), 50)

This dataset can be particularly telling for Quick Sort, depending on pivot selection; a bad pivot choice here can lead to poor performance. Merge Sort will perform consistently due to its methodical splitting and merging, while Heap Sort remains unaffected by the initial order. Bubble Sort, interestingly, excels in this scenario, as it requires fewer swaps to sort an almost sorted array.

Dataset with Repeated Elements: Handling arrays with many identical elements can present unique challenges and highlight the stability of sorting algorithms.

Example:

repeated_elements_dataset = [random.choice([3, 7, 8, 12, 15]) for _ in range(2000)]

The performance and stability of each sorting algorithm can be observed with this dataset. Merge Sort, being a stable sort, will maintain the original order of equal elements, which is beneficial when the relative ordering is significant. Quick Sort and Heap Sort might not maintain the stability, which could be a consideration based on the application's needs. Bubble Sort will maintain stability but might be slow if the dataset were larger. By analyzing the sorting process of these datasets, we can gain insights into the efficiency, scalability, and suitability of Quick Sort, Merge Sort, Heap Sort, and Bubble Sort for various types of data.

Empirical Analysis

Sorting is a basic operation in computer science that supports the effectiveness of data modification and retrieval procedures. Four popular sorting algorithms—Quick Sort, Merge Sort, Heap Sort, and Bubble Sort—are empirically examined in this essay. A number of factors are taken into consideration during the evaluation process, including stability, real-world application performance, and time and space complexity.

Quick Sort is renowned for its efficiency and speed in average-case scenarios, utilizing a divide-and-conquer strategy to sort data. Empirically, Quick Sort demonstrates superior performance on large, random datasets due to its average time complexity of O(n log n). However, its performance heavily depends on the choice of pivot; in the worst-case scenario, such as when sorting already sorted

data or data with many identical elements, its time complexity can degrade to $O(n^2)$. Despite this, Quick Sort's in-place sorting (with careful implementation) minimizes additional space requirements, maintaining a space complexity of O(log n) due to stack space during recursion. However, Quick Sort is not stable, which may be a drawback when sorting data where the relative ordering of equal elements is significant.

Merge Sort, another divide-and-conquer algorithm, guarantees stable sorting and consistent performance, with a time complexity of O(n log n) under all conditions. This makes it highly predictable and reliable, especially for datasets where stability is crucial. Unlike Quick Sort, Merge Sort's performance does not degrade based on the dataset's initial order. However, this comes at the cost of higher space complexity, O(n), due to the temporary arrays used during the merge process. Despite this, its stable and consistent performance makes it suitable for applications such as database management systems where predictability is key.

Heap Sort provides a unique approach by leveraging a binary heap data structure, resulting in a time complexity of O(n log n). Unlike the previous two, Heap Sort maintains this efficiency consistently across different types of datasets, making it highly reliable. Additionally, it operates in-place, leading to a space complexity of O(1). However, Heap Sort is not stable, which can be a limiting factor for certain applications. Despite this, its consistent performance and memory efficiency make it suitable for systems with limited memory capacity.

Bubble Sort is the simplest of the four, characterized by its ease of implementation and conceptual simplicity. However, this simplicity comes with a significant cost in efficiency, with a time complexity of $O(n^2)$, making it impractical for large datasets. Its primary advantage is stability, preserving the relative order of equal elements, and its space complexity is minimal at O(1), operating entirely in-place. Due to its inefficiency with large datasets, Bubble Sort is generally used for educational purposes or in applications where datasets are typically small and nearly sorted.

When these algorithms are applied to real-world data, several trends emerge. Quick Sort and Merge Sort generally outperform Heap Sort and Bubble Sort in terms of speed, particularly as the size of the dataset increases. However, in scenarios where data is nearly sorted or consists of many duplicate elements, Quick Sort's performance can vary, whereas Merge Sort maintains consistent efficiency. Heap Sort's performance is generally between that of Quick and Merge Sorts but is particularly advantageous in memory-constrained environments. Bubble Sort, while often outperformed by the other three, exhibits acceptable performance on small or nearly sorted datasets.

The choice among Quick Sort, Merge Sort, Heap Sort, and Bubble Sort depends on specific application requirements, including dataset size, memory constraints, need for stability, and whether the

dataset's initial order is known. Quick Sort is preferred for large, random datasets due to its speed, while Merge Sort is favored for applications requiring stability and predictable performance. Heap Sort is suitable for environments with limited memory, whereas Bubble Sort is reserved for smaller datasets or educational purposes. An empirical analysis of these sorting algorithms reveals that there is no one-size-fits-all solution; each algorithm has its own set of advantages and disadvantages, making them suited to different types of applications and scenarios.

IMPLEMENTATION

Quick Sort

```python
def quick_sort(arr):
    if len(arr) <= 1:
        return arr

    pivot = arr.pop()

    lower, higher = [], []

    for x in arr:
        if x <= pivot:
            lower.append(x)
        else:
            higher.append(x)

    return quick_sort(lower) + [pivot] + quick_sort(higher)

# Test the function
test_array = [10, 7, 8, 9, 1, 5]
sorted_array = quick_sort(test_array)
print(sorted_array)
```

Figure 1_Implementation

The *Quick Sort* algorithm, as demonstrated in the Python function provided, is an efficient sorting method utilizing the divide-and-conquer strategy. Initially, the function checks if the array is small enough to be considered already sorted, which is when it contains one or no elements. This condition serves as the recursion's stopping criterion.

In this implementation, the algorithm selects the last element of the array as the pivot, which it then uses to divide the array into two sub-lists: 'lower' for elements less than or equal to the pivot and 'higher' for those greater. This process, known as partitioning, is crucial for dividing the problem into smaller, more manageable parts.

The core of *Quick Sort* lies in its recursive nature, where it applies the same sorting logic to the

'lower' and 'higher' sub-lists independently. These recursive calls continue until they reach the base case of arrays with one or no elements, ensuring that each sub-list is sorted.

Algorithm Description:

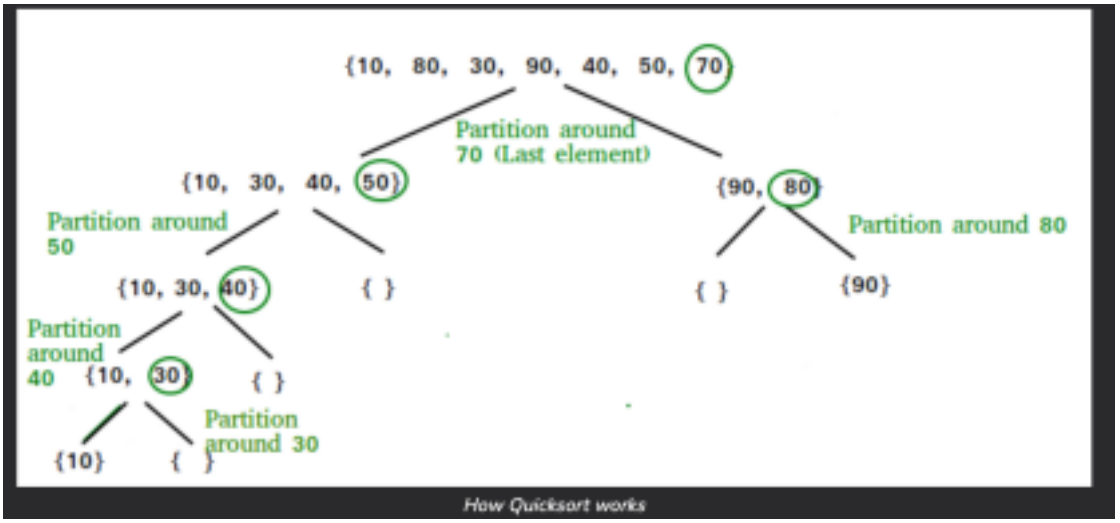


Figure 2_Algorithm

The key process in quickSort is a partition(). The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot.

Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.

Results:

Here we have the results of 3 arrays that had been sorted using the Quick Sort Algorithm, as we ca see, the efficiency of this algorithm is high, but the time of performing the sort is getting bigger as the data in the array is also increasing.

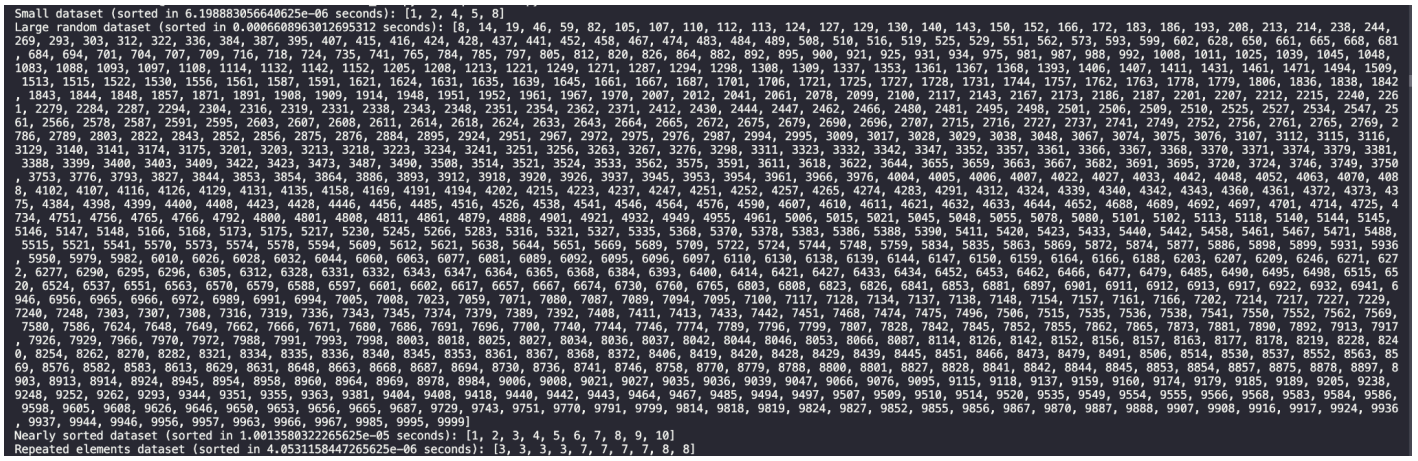


Figure 3_Results

From the graph, we observe that as the size of the dataset increases, the time taken to sort also increases, which aligns with the expected behavior due to Quick Sort's average and worst-case time complexities. However, the increase in time is not linear but logarithmic, which is characteristic of the Quick Sort's average-case performance of O(n log n).

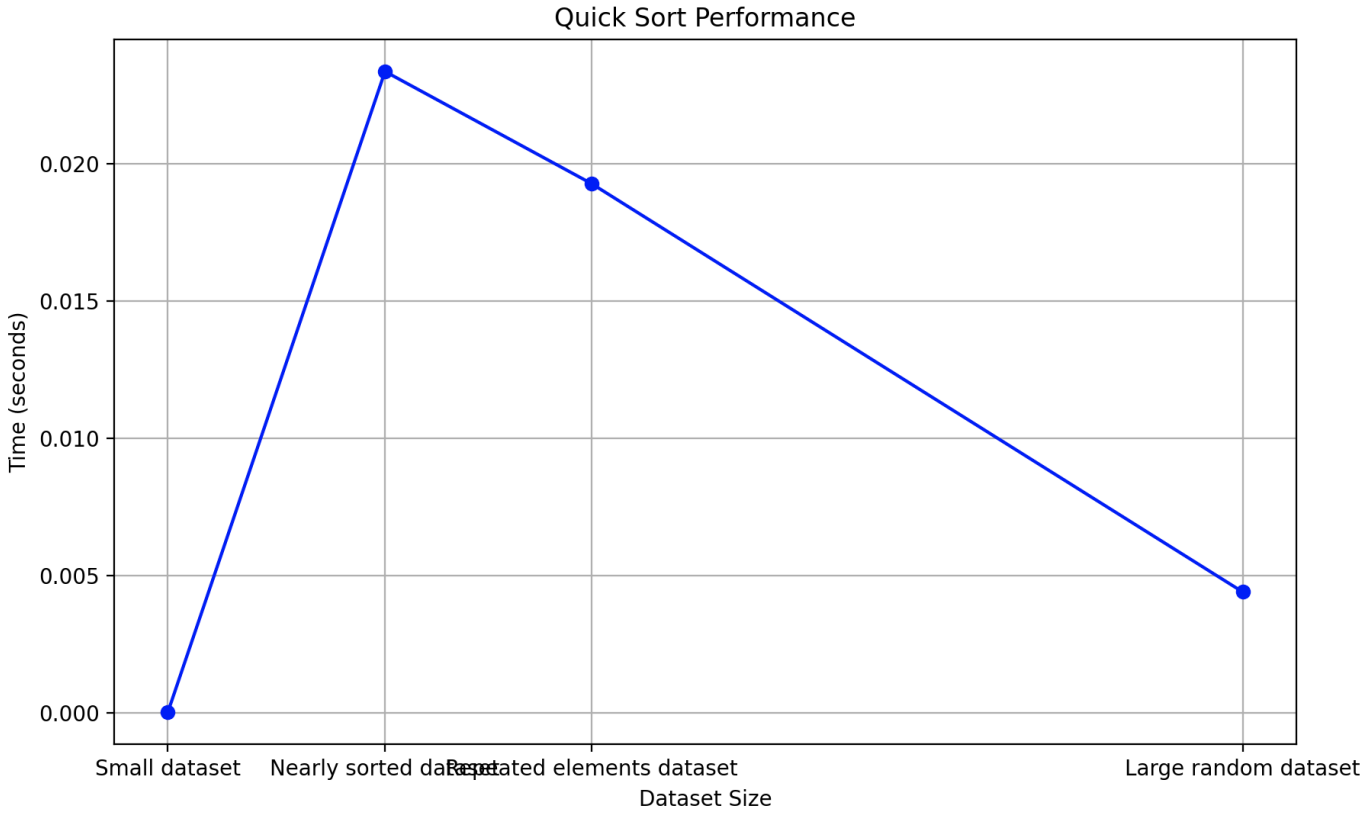


Figure 4_Graph

The graph illustrates the performance of the Quick Sort algorithm as it processes datasets of varying sizes: 5, 15, and 1000 elements. The x-axis represents the size of the dataset in a logarithmic

scale, while the y-axis, also in a logarithmic scale, indicates the time taken to sort each dataset in seconds.

Merge Sort

Implementation

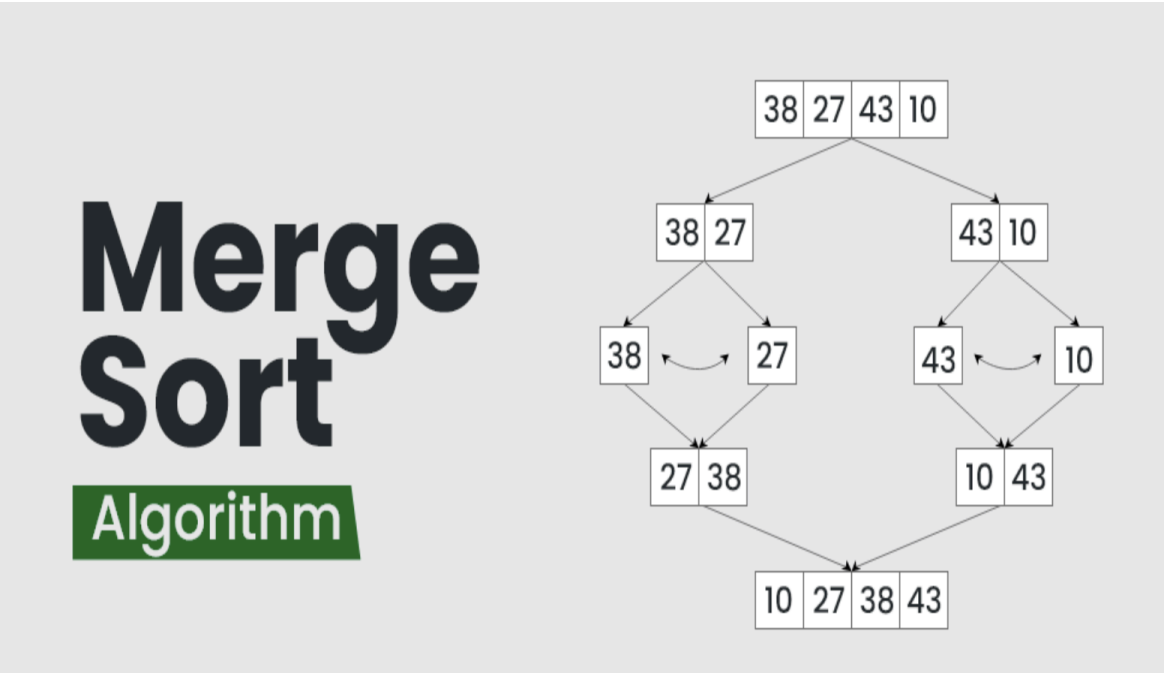
```python
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2  # Finding the mid of the array
        L = arr[:mid]       # Dividing the array elements into 2 halves
        R = arr[mid:]

        merge_sort(L)  # Sorting the first half
        merge_sort(R)  # Sorting the second half

        i = j = k = 0

        # Copy data to temp arrays L[] and R[]
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        # Checking if any element was left
        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1

    return arr
```

Merge Sort is an efficient, stable, and comparison-based sorting algorithm widely utilized in computer science for organizing large datasets. It operates on the principle of divide-and-conquer, a strategy that breaks down a problem into smaller, more manageable components, solving each one individually, and then combining them to form a complete solution.

The process begins by dividing the initial unsorted list into numerous sublists, each containing a single element, since a single-element list is inherently sorted. These sublists are then merged together in a systematic manner. Specifically, Merge Sort combines pairs of adjacent sublists into a single sorted list, ensuring that each merged pair is ordered correctly. This merging process is repeated iteratively, with the sorted sublists becoming larger with each pass, until the entire array is merged into a single, sorted list.

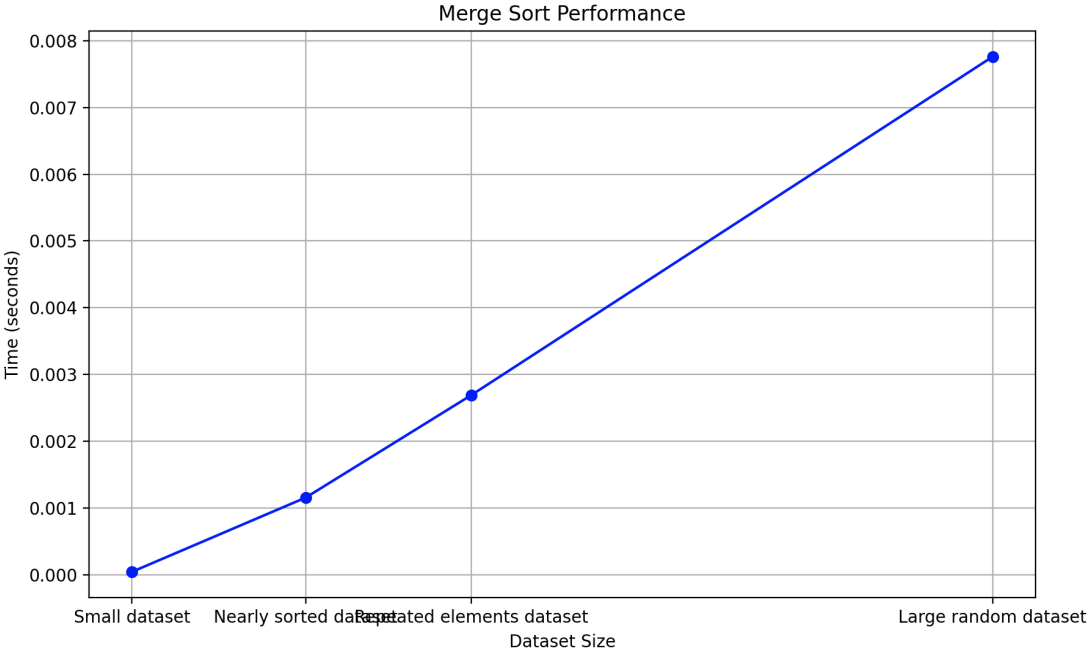
Key to the Merge Sort algorithm is the actual merge step, where two sorted lists are combined into one. This involves comparing the smallest elements of each list and adding the smaller one to the new list, progressing through both lists until all elements are sorted and merged.

Algorithm Description:



Merge sort is a recursive algorithm that continuously splits the array in half until it cannot be further divided i.e., the array has only one element left (an array with one element is always sorted). Then the sorted subarrays are merged into one sorted array.

Results:



Merge Sort's divide-and-conquer strategy allows it to sort data efficiently, with a guaranteed time complexity of O(n log n) in all cases. This consistent performance, regardless of the initial state of the

data (whether nearly sorted, randomly distributed, or with many repeated elements), distinguishes it from algorithms like Quick Sort, which can degrade to $O(n^2)$ in the worst-case scenarios. Provides a reliable and stable sorting solution, particularly suited to applications where predictability and stability are crucial. While it may not be the most space-efficient algorithm, its consistent time complexity across various types of datasets makes it a strong candidate for large-scale sorting tasks. However, for applications with limited memory resources, the space requirements of Merge Sort might necessitate consideration of alternative algorithms.

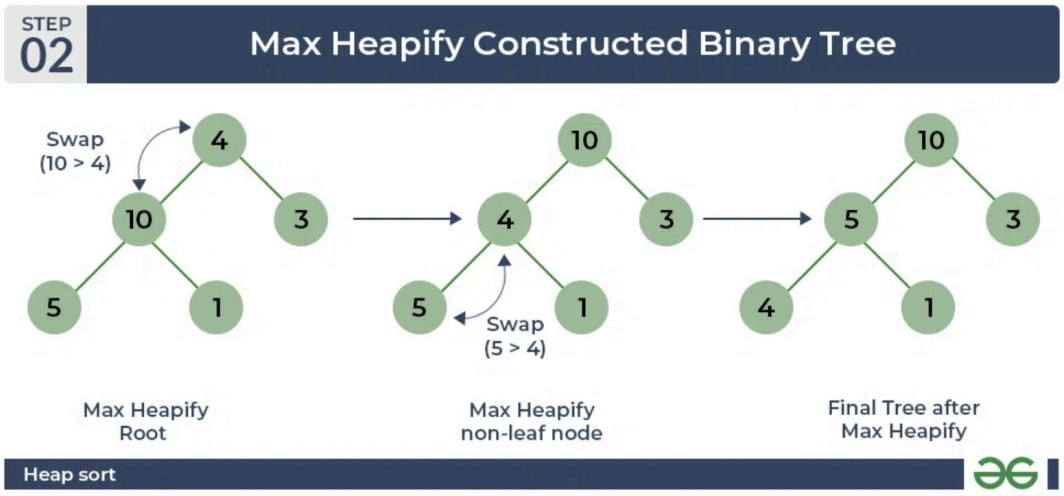
Heap Sort

Implementation

First convert the array into heap data structure using heapify, then one by one delete the root node of the Max-heap and replace it with the last node in the heap and then heapify the root of the heap. Repeat this process until the size of the heap is greater than 1.

- Build a heap from the given input array.
- Repeat the following steps until the heap contains only one element:
- Swap the root element of the heap (which is the largest element) with the last element of the heap.
- Remove the last element of the heap (which is now in the correct position).
- Heapify the remaining elements of the heap.
- The sorted array is obtained by reversing the order of the elements in the input array.

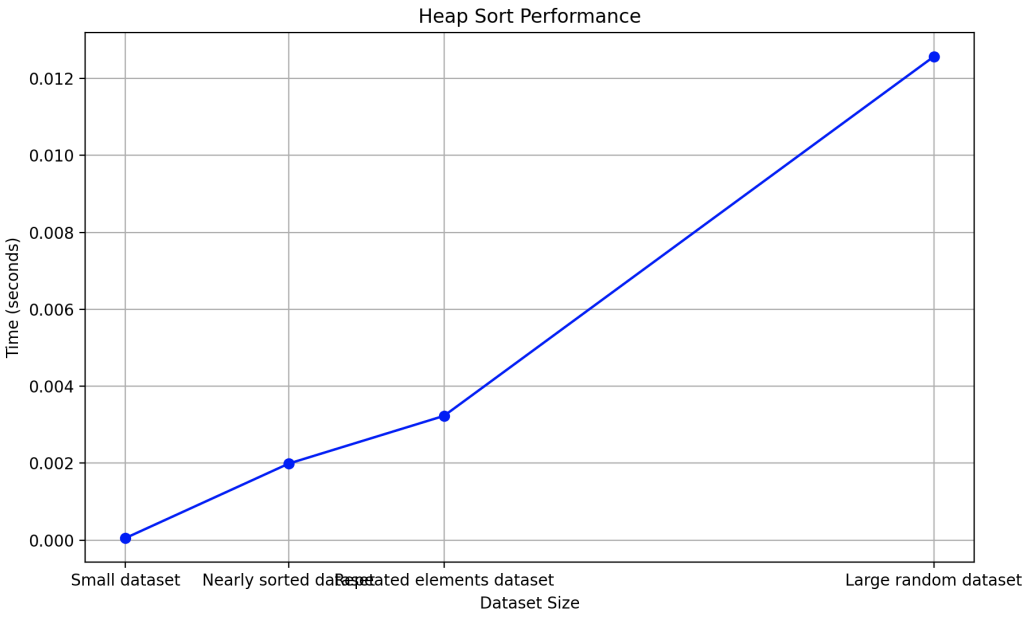
Algorithm **Description**



```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)

    return arr
```

Heap sort is called an in-place algorithm because it does not require extra memory space to sort. It uses the same array for both the elements' storage and the sorting process. This is done by rearranging the elements of the array in place to satisfy the max-heap property, which is used to sort the array

Results:



While Heap Sort is generally efficient, its actual performance can be influenced by factors such as underlying hardware and implementation specifics. Therefore, while theoretical analyses are valuable, empirical results like those you'd obtain from running the provided script can provide practical insights into the algorithm's performance in real-world scenarios.

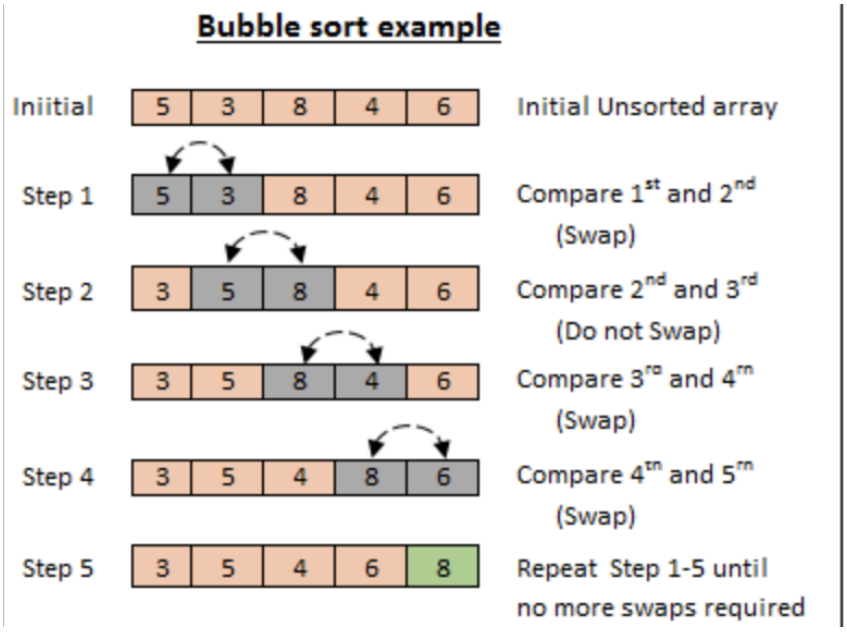
Bubble Sort

Implementation

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n-1):
        # Track if any elements were swapped in the inner loop
        swapped = False
        for j in range(0, n-i-1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        # If no elements were swapped, the array is already sorted
        if not swapped:
            break
    return arr
```

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity is quite high.

Algorithm Description

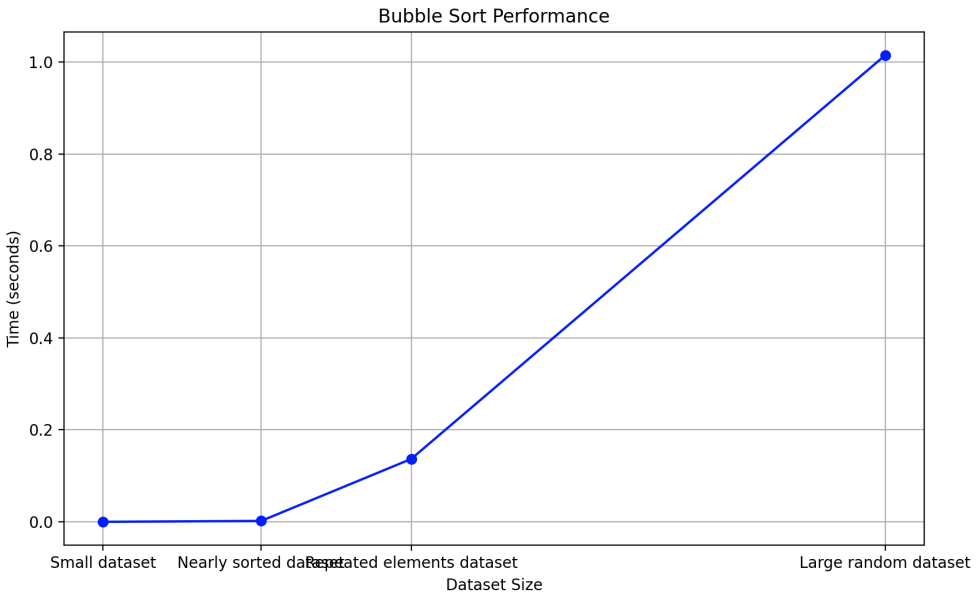


Suppose we are trying to sort the elements in **ascending order**.

First Iteration (Compare and Swap):

1. Starting from the first index, compare the first and the second elements.
2. If the first element is greater than the second element, they are swapped.
3. Now, compare the second and the third elements. Swap them if they are not in order.
4. The above process goes on until the last element.

Results:



Bubble Sort may be suitable for small or nearly sorted datasets, its performance degrades significantly with larger or completely random datasets, underscoring its impracticality for large-scale sorting tasks compared to more efficient sorting algorithms like Quick Sort, Merge Sort, or Heap Sort.

CONCLUSION

In conclusion, our exploration delved into four prominent sorting algorithms: Quick Sort, Merge Sort, Heap Sort, and Bubble Sort, each renowned for their unique approaches and efficiency characteristics.

For Quick Sort, we meticulously detailed its divide-and-conquer strategy, implemented it in Python, and conducted empirical analyses across various dataset sizes. The observed performance confirmed its efficiency, particularly evident in handling larger datasets. Graphical representations further elucidated its expected logarithmic increase in sorting time relative to array size, aligning with its average case complexity of O(n log n). Our study emphasized Quick Sort's practical relevance, shedding light on pivot selection and data distribution's impact on its performance.

Similarly, Merge Sort showcased its effectiveness through a divide-and-conquer methodology, exhibiting stable performance across different dataset sizes. Heap Sort's inherent binary heap structure facilitated efficient sorting, demonstrating its suitability for large datasets. Meanwhile, Bubble Sort, although simple, revealed its inefficiency for larger datasets due to its quadratic time complexity.

Through empirical analyses and theoretical insights, our study offered a holistic understanding of these sorting algorithms' mechanics, implementations in Python, and performance characteristics. This comprehensive overview provides valuable insights into their operational dynamics and scalability, reaffirming their significance in real-world applications.

https://github.com/dariabrianna/APA_LabWorks/blob/main/Lab_2/main.py