Ministry of Education, Culture and Research of the Republic of Moldova Technical University of Moldova
Department of Software and Automation Engineering

Laboratory work 2:

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

Elaborated:
st. gr. FAF-223 Tofan Liviu

Verified:
asist. univ. Fiștic Cristofor

Chişinău – 2024

TABLE OF CONTENTS

ALGORITHM ANALYSIS

Objective:

1. Study of the method divide et impera
2. Analysis and implementation of algorithms based on the divide et impera method

Tasks:

1. Implement 4 sorting algorithms.
2. Decide properties of input format that will be used for algorithm analysis;
3. Decide the comparison metric for the algorithms;
4. Analyze empirically the algorithms;
5. Present the results of the obtained data;
6. Deduce conclusions of the laboratory.

Theoretical Notes:

An alternative to mathematical analysis of the complexity of sorting algorithms is empirical analysis. This approach can be valuable for various purposes, including obtaining preliminary insights into the complexity class of a sorting algorithm, comparing the efficiency of different sorting algorithms tackling the same problem, evaluating the efficiency of multiple implementations of a specific sorting algorithm, or gathering information on the efficiency of implementing a sorting algorithm on a particular computing environment. In the empirical analysis of an algorithm, the following steps are usually followed:

1. *Establishing the Purpose of the Analysis:*

Define the specific goals and objectives of the empirical analysis. This could include understanding the runtime behaviour, comparing performance, or assessing the practical efficiency of sorting algorithms.

2. *Choosing Efficiency Metrics:*

Select appropriate efficiency metrics for sorting algorithms. Common metrics include the number of key comparisons, swaps, or the total execution time for sorting a given dataset.

3. *Defining Input Data Properties:*

Identify and establish the properties of the input data relevant to the analysis. This may involve specifying the size of the dataset or considering specific characteristics of the input elements.

4. *Algorithm Implementation:*

Implement the sorting algorithm in a programming language of choice. Ensure that the implementation is correct and adheres to the chosen algorithm's logic.

5. *Generating Multiple Sets of Input Data:*

Create diverse sets of input data to cover a range of scenarios, such as best-case, average-case, and worst-case inputs. Vary the size and characteristics of the datasets.

6. *Running the Program for Each Input Data Set:*

Execute the sorting algorithm for each generated dataset. Record relevant information during the execution, such as the number of comparisons, swaps, or the execution time.

7. *Analyzing Obtained Data:*

Analyze the collected data to draw meaningful conclusions. This may involve calculating synthetic quantities such as mean, standard deviation, etc., or creating graphs with pairs of points representing the problem size and the chosen efficiency measure.

The choice of efficiency measure depends on the specific goals of the analysis. If the aim is to understand the theoretical complexity class or validate theoretical estimates, counting the number of key operations (comparisons or swaps) might be appropriate. On the other hand, if the focus is on assessing real-world performance, measuring the execution time is more relevant. After running the program with the test datasets, the results can be recorded and analyzed, providing insights into the practical efficiency and behaviour of the sorting algorithm under different scenarios. The choice of the efficiency measure depends on the purpose of the analysis. If, for example, the aim is to obtain information on the complexity class or even checking the accuracy of a theoretical estimate then it is appropriate to use the number of operations performed. But if the goal is to assess the behaviour of the implementation of an algorithm then execution time is appropriate. After the execution of the program with the test data, the results are recorded and, for the purpose of the analysis, either synthetic quantities (mean, standard deviation, etc.) are calculated or a graph with appropriate pairs of points (i.e. problem size, efficiency measure) is plotted. Introduction: Sorting algorithms are fundamental tools in the realm of computer science, playing a pivotal role in organizing and arranging data in a systematic order. The primary objective of sorting is to enhance the accessibility and searchability of information, facilitating efficient retrieval and manipulation of data sets. As the backbone of numerous applications and systems, sorting algorithms are omnipresent in everyday computing tasks. The ubiquity of sorting algorithms spans a spectrum of applications, from simple list organization to complex database management. Tasks as varied as arranging a list of names alphabetically, searching for specific elements efficiently, or optimizing the performance of search algorithms heavily rely on the effectiveness of sorting techniques. The importance of sorting algorithms extends beyond basic data arrangement; it significantly impacts the efficiency and responsiveness of various software systems. Whether in databases, information retrieval, or optimizing resource utilization in memory, the choice and implementation of sorting algorithms can markedly influence the overall performance of computational processes. In essence, sorting algorithms serve as the bedrock for a wide array of applications, contributing to the seamless functioning of everyday computing tasks. As we delve into the intricacies of sorting algorithms, understanding their principles and nuances becomes imperative for any computer scientist or programmer aiming to design robust and efficient systems.

Introduction:

Sorting algorithms are fundamental tools in the realm of computer science, playing a pivotal role in organizing and arranging data in a systematic order. The primary objective of sorting is to enhance the accessibility and searchability of information, facilitating efficient retrieval and manipulation of data sets. As the backbone of numerous applications and systems, sorting algorithms are omnipresent in everyday computing tasks. The ubiquity of sorting algorithms spans a spectrum of applications, from simple list organization to complex database management. Tasks as varied as arranging a list of names alphabetically, searching for specific elements efficiently, or optimizing the performance of search algorithms heavily rely on the effectiveness of sorting techniques. The importance of sorting algorithms extends beyond basic data arrangement; it significantly impacts the efficiency and responsiveness of various software systems. Whether in databases, information retrieval, or optimizing resource utilization in memory, the choice and implementation of sorting algorithms can markedly influence the overall performance of computational processes. In essence, sorting algorithms serve as the bedrock for a wide array of applications, contributing to the seamless functioning of everyday computing tasks. As we delve into the intricacies of sorting algorithms, understanding their principles and nuances becomes imperative for any computer scientist or programmer aiming to design robust and efficient systems.

Comparison Metric:

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

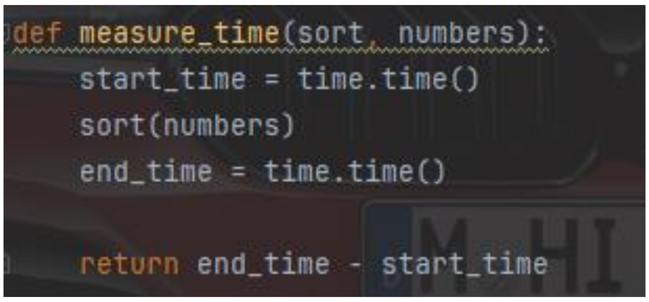
Input Format:

As input, each algorithm will receive lists of varying lengths, including 1000, 100000, and 1000000 elements. These lists consist of positive integer numbers. For plotting on a graph, I create lists starting with 5000 elements, incrementing by 5000, and reaching up to 100000. This range covers diverse scenarios and facilitates a comprehensive analysis. This is done with integers, floats and negative numbers offering a broad spectrum of input data to evaluate the sorting algorithm's performance across different data types and sizes.

IMPLEMENTATION

All four algorithms will be implemented in their naïve 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 memory of the device used.

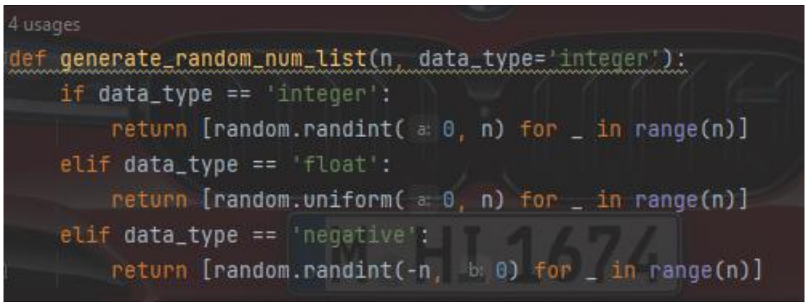
For measuring time I created the follow function:



```
def measure_time(sort, numbers):
    start_time = time.time()
    sort(numbers)
    end_time = time.time()

    return end_time - start_time
```

Figure 1. Function to measure time execution for sorting algorithm.

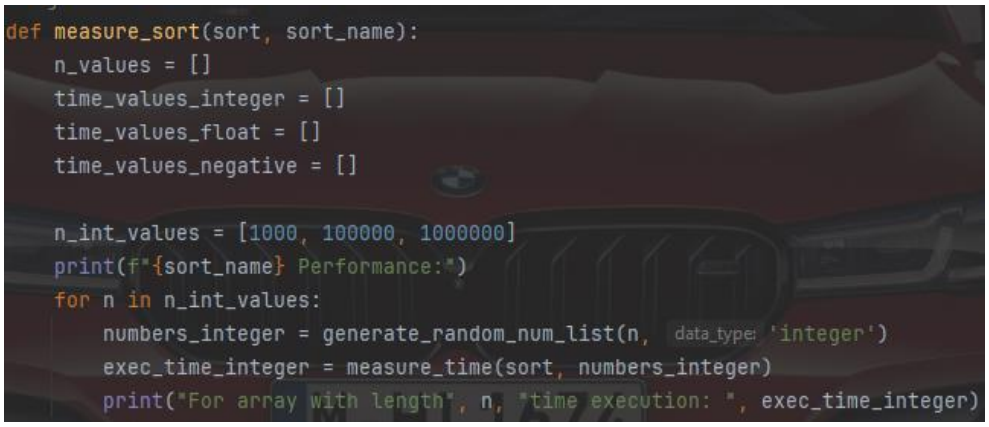
For generating arrays with numbers of different types I created the follow function:



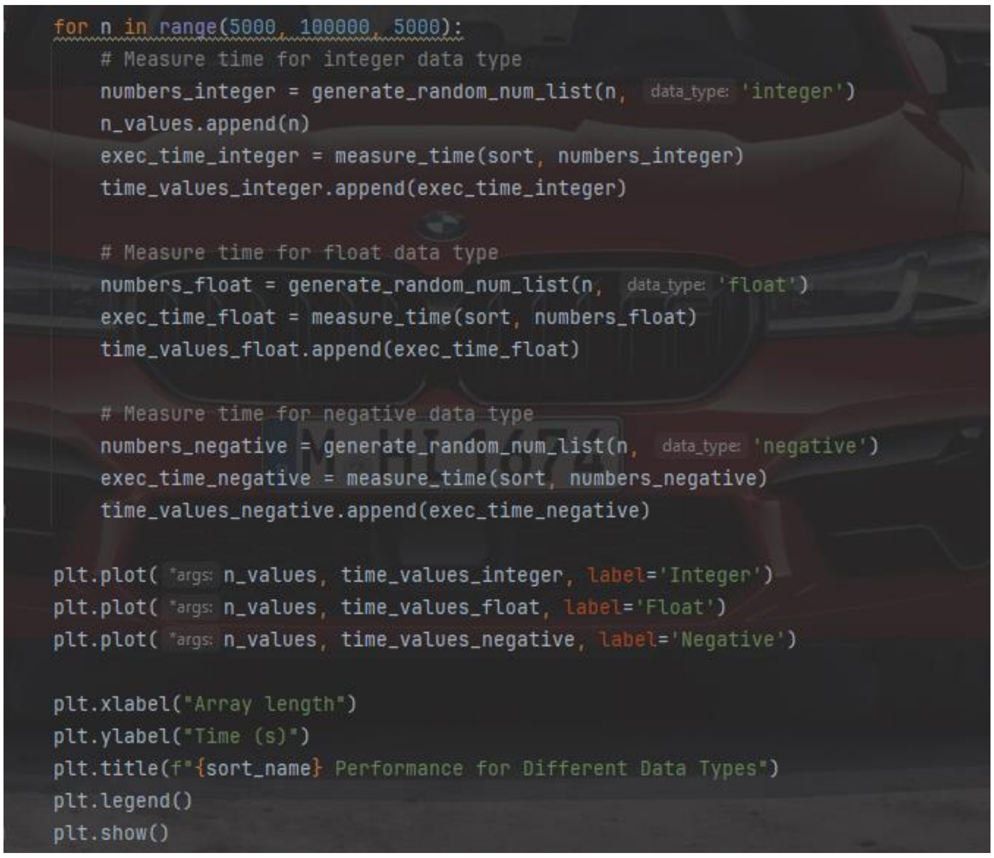
```
4 usages
def generate_random_num_list(n, data_type='integer'):
    if data_type == 'integer':
        return [random.randint( a: 0, n) for _ in range(n)]
    elif data_type == 'float':
        return [random.uniform( a: 0, n) for _ in range(n)]
    elif data_type == 'negative':
        return [random.randint(-n,  b: 0) for _ in range(n)]
```

Figure 2. Function to generate arrays of different data types and lengths numbers.

And to compare time execution for each sort, and plotting data I have the follow function:



```
def measure_sort(sort, sort_name):
    n_values = []
    time_values_integer = []
    time_values_float = []
    time_values_negative = []

    n_int_values = [1000, 100000, 1000000]
    print(f"{sort_name} Performance:")
    for n in n_int_values:
        numbers_integer = generate_random_num_list(n,  data_type: 'integer')
        exec_time_integer = measure_time(sort, numbers_integer)
        print("For array with length", n, "time execution: ", exec_time_integer)
```



```
    for n in range(5000, 100000, 5000):
        # Measure time for integer data type
        numbers_integer = generate_random_num_list(n,  data_type: 'integer')
        n_values.append(n)
        exec_time_integer = measure_time(sort, numbers_integer)
        time_values_integer.append(exec_time_integer)

        # Measure time for float data type
        numbers_float = generate_random_num_list(n,  data_type: 'float')
        exec_time_float = measure_time(sort, numbers_float)
        time_values_float.append(exec_time_float)

        # Measure time for negative data type
        numbers_negative = generate_random_num_list(n,  data_type: 'negative')
        exec_time_negative = measure_time(sort, numbers_negative)
        time_values_negative.append(exec_time_negative)

    plt.plot( *args: n_values, time_values_integer, label='Integer')
    plt.plot( *args: n_values, time_values_float, label='Float')
    plt.plot( *args: n_values, time_values_negative, label='Negative')

    plt.xlabel("Array length")
    plt.ylabel("Time (s)")
    plt.title(f"{sort_name} Performance for Different Data Types")
    plt.legend()
    plt.show()
```

Figure 3. Function to plot results.

Quick Sort:

QuickSort is a sorting algorithm based on the Divide and Conquer algorithm 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. 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.

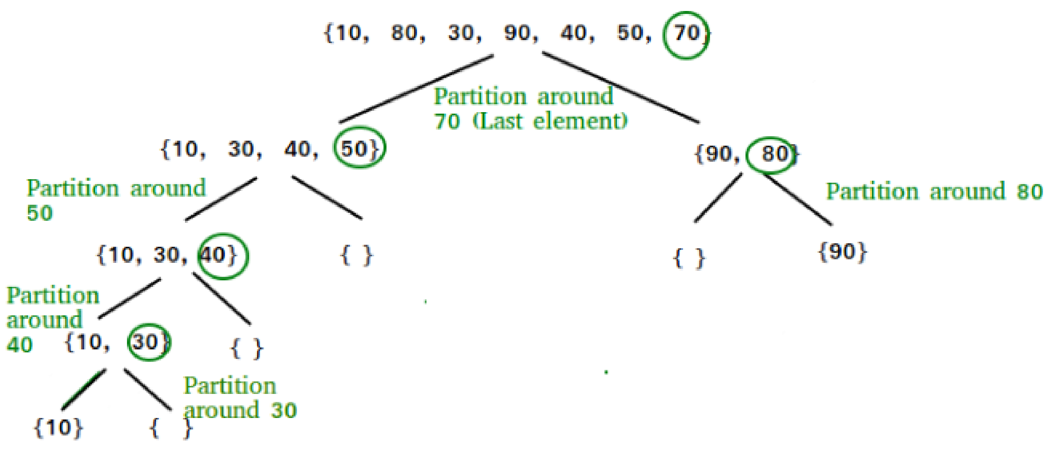


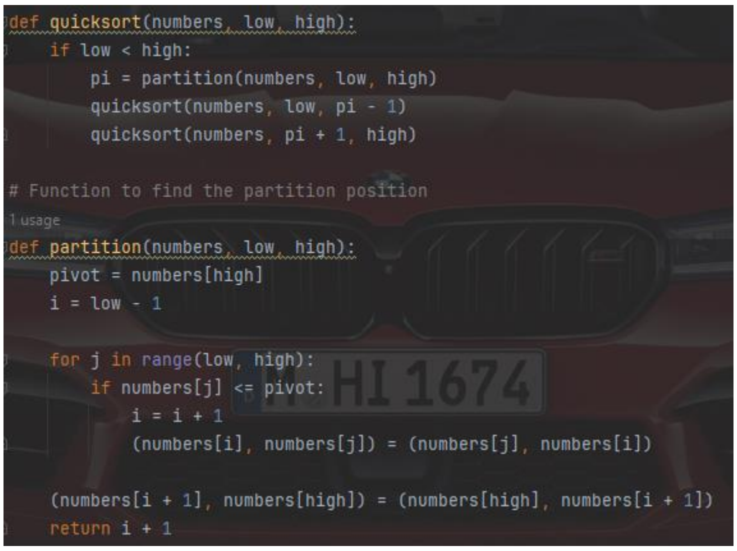
Figure 4. How Quicksort works.

Algorithm Description:

The naive recursive Quicksort method follows the algorithm as shown in the next pseudocode:

```
quicksort(arr, low, high):
    if low < high:
        // Partition the array
        pivotIndex = partition(arr, low, high)

        // Recursively sort the sub-arrays
        quicksort(arr, low, pivotIndex - 1)
        quicksort(arr, pivotIndex + 1, high)

partition(arr, low, high):
    // Choose the rightmost element as the pivot
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        // If the current element is smaller than or equal to the pivot
        if arr[j] <= pivot:
            // Swap arr[i+1] and arr[j]
            swap(arr, i+1, j)
            i = i + 1

    // Swap arr[i+1] and arr[high] to place the pivot in the correct position
    swap(arr, i+1, high)

    // Return the index of the pivot element
    return i+1

swap(arr, i, j):
    // Utility function to swap elements in an array
    temp = arr[i]
    arr[i] = arr[j]
    arr[j] = temp
```

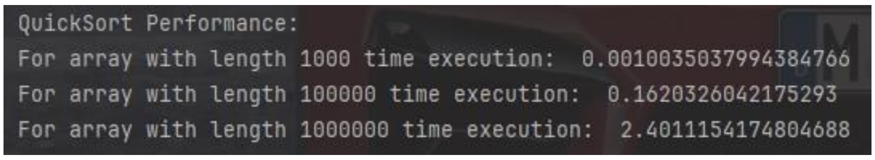
Implementation:



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

# Function to find the partition position
1 usage
def partition(numbers, low, high):
    pivot = numbers[high]
    i = low - 1

    for j in range(low, high):
        if numbers[j] <= pivot:
            i = i + 1
            (numbers[i], numbers[j]) = (numbers[j], numbers[i])

    (numbers[i + 1], numbers[high]) = (numbers[high], numbers[i + 1])
    return i + 1
```

Figure 5. Quicksort in Python.

Results:



```
QuickSort Performance:
For array with length 1000 time execution:  0.0010035037994384766
For array with length 100000 time execution:  0.1620326042175293
For array with length 1000000 time execution:  2.4011154174804688
```

Figure 6. Quicksort results for large arrays with integer numbers.

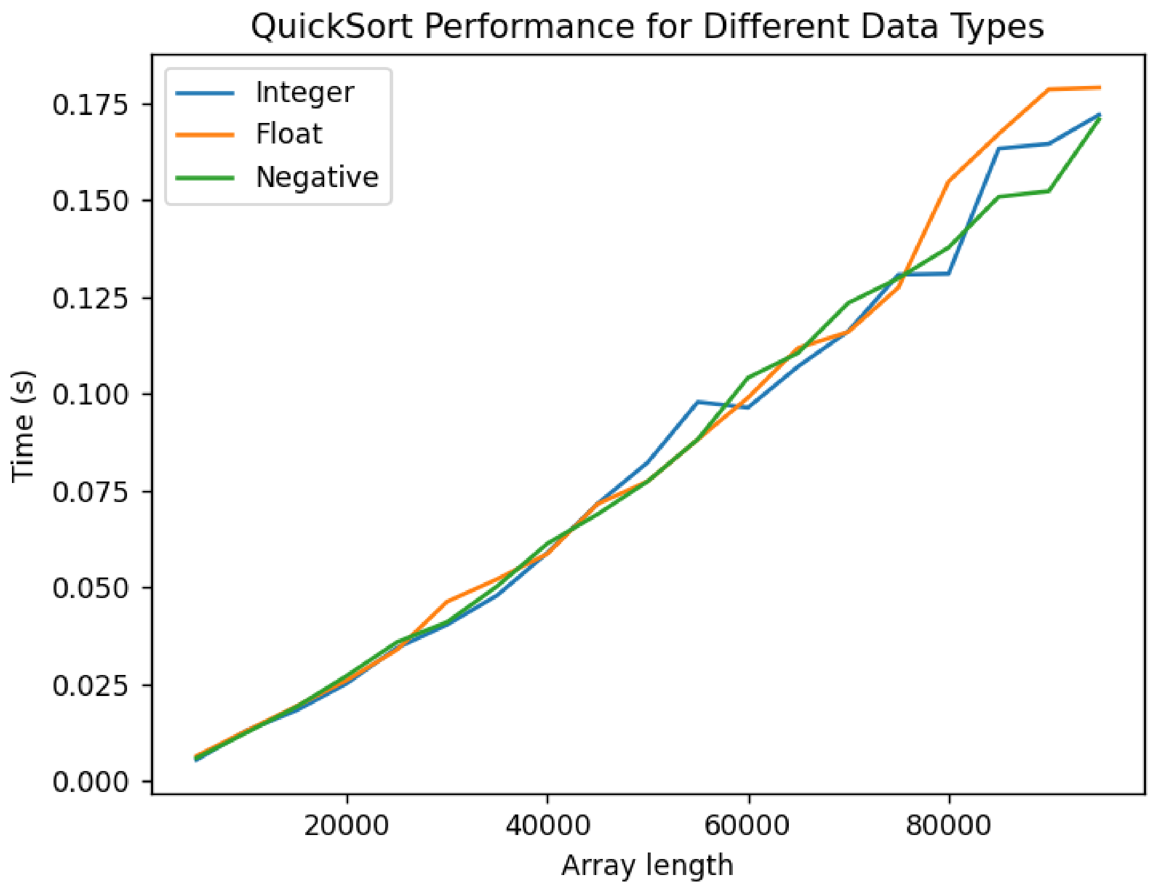


Figure 7. Quicksort results for large arrays with different data types.

As observed in the above images, QuickSort proves to be a highly efficient sorting algorithm. It demonstrated remarkable speed, sorting a million-element integer array in just 2.4 seconds. Furthermore, the plotted graph reveals that the execution time is consistently similar for all three data types (integer, float, and negative), with a slight advantage for floats, being marginally faster by around 100k elements. This underscores the versatility and effectiveness of QuickSort across various data scenarios.

Time Complexity:

Best Case: T (N log (N)) The best-case scenario for quicksort occur when the pivot chosen at the each step divides the array into roughly equal halves. In this case, the algorithm will make balanced partitions, leading to efficient Sorting.

Average Case: T(N log (N)) Quicksort's average-case performance is usually very good in practice, making it one of the fastest sorting Algorithm.

Worst Case: T(N2) The worst-case Scenario for Quicksort occur when the pivot at each step consistently results in highly unbalanced partitions. When the array is already sorted and the pivot is always chosen as the smallest or largest element. To mitigate the worst-case Scenario, various techniques are used such as choosing a good pivot (e.g., median of three) and using Randomized algorithm (Randomized Quicksort) to shuffle the element before sorting.

Auxiliary Space: T(1), if we don't consider the recursive stack space. If we consider the recursive stack space then, in the worst case quicksort could make O(N).

Advantages of Quick Sort:

- It is a divide-and-conquer algorithm that makes it easier to solve problems.
- It is efficient on large data sets.
- It has a low overhead, as it only requires a small amount of memory to function.

Disadvantages of Quick Sort:

- It has a worst-case time complexity of T(N2), which occurs when the pivot is chosen poorly.
- It is not a good choice for small data sets.
- It is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivot's position (without considering their original positions).

Merge Sort:

Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final 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.

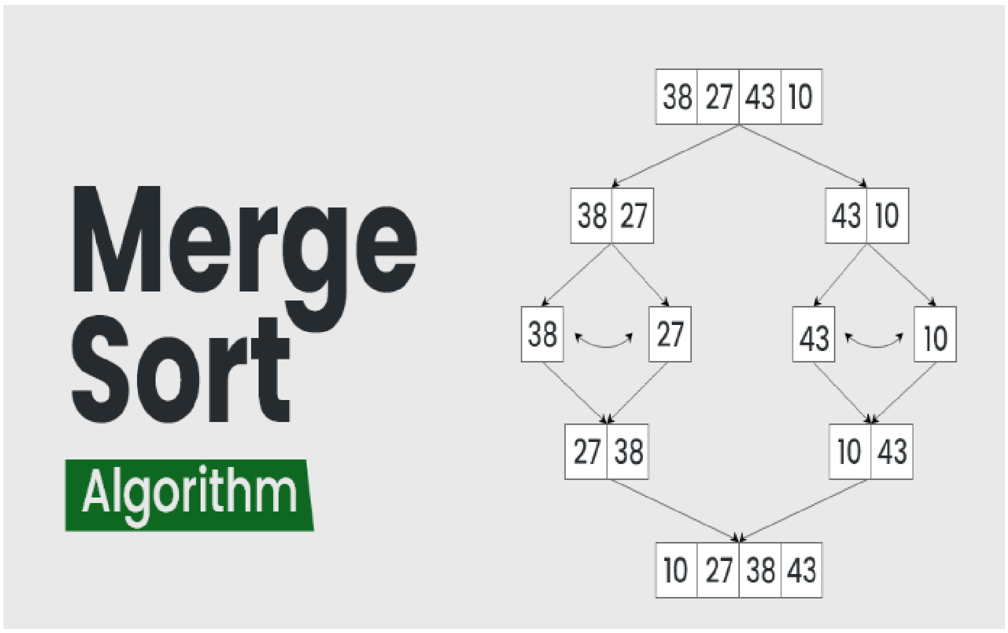


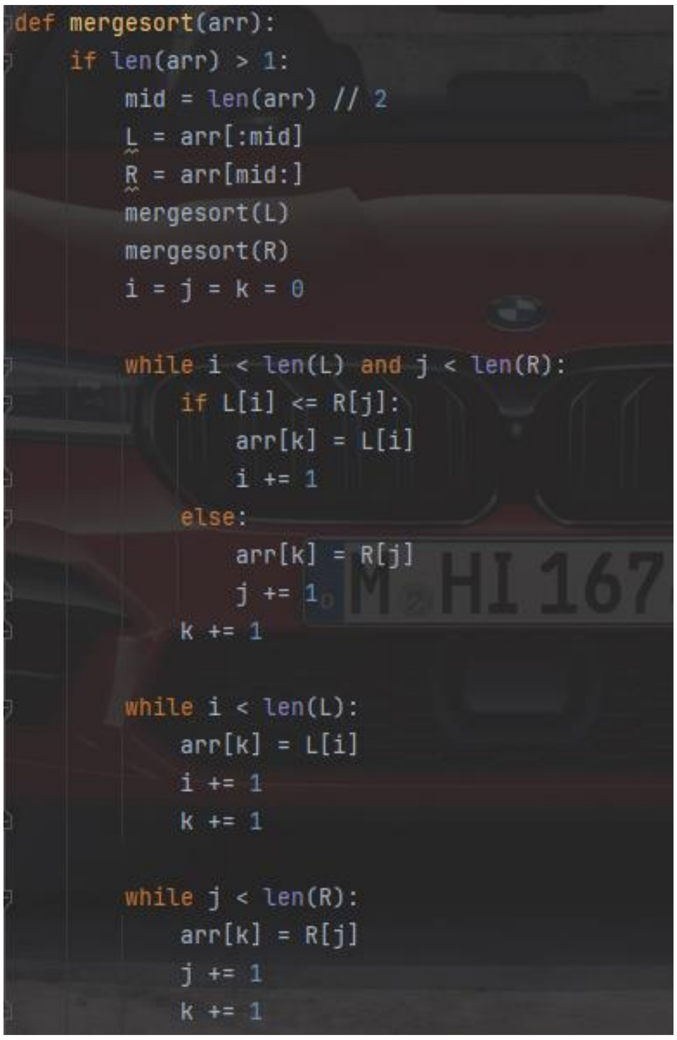
Figure 8. How MergeSort works.

Algorithm Description:

The naive recursive MergeSort method follows the algorithm as shown in the next pseudocode:

```
mergesort(arr):
    if length of arr > 1:
        mid = length of arr // 2
        // Divide the array into two halves
        L = arr[0:mid]
        R = arr[mid:end]
        // Recursively sort the left and right halves
        mergesort(L)
        mergesort(R)

        i = j = k = 0

        // Merge the sorted halves
        while i < length of L and j < length of R:
            if L[i] <= R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        // Copy any remaining elements of L and R, if any
        while i < length of L:
            arr[k] = L[i]
            i += 1
            k += 1
        while j < length of R:
            arr[k] = R[j]
            j += 1
            k += 1
```

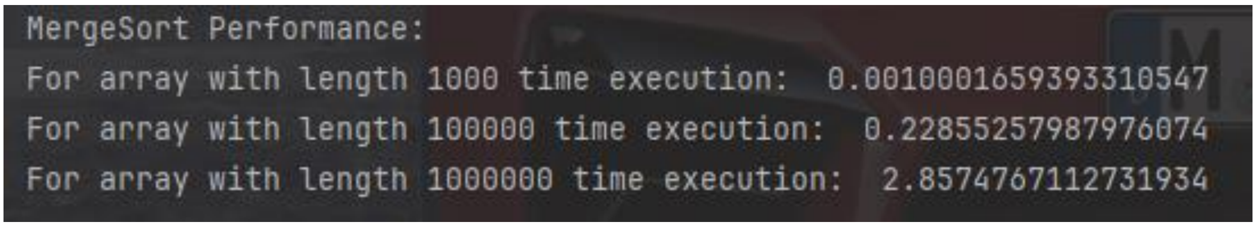
Implementation:



```python
def mergesort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]
        mergesort(L)
        mergesort(R)
        i = j = k = 0

        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

        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

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

Figure 9. MergeSort in Python.

Results:



```
MergeSort Performance:
For array with length 1000 time execution:  0.0010001659393310547
For array with length 100000 time execution:  0.22855257987976074
For array with length 1000000 time execution:  2.8574767112731934
```

Figure 10. MergeSort results for large arrays with integer numbers.

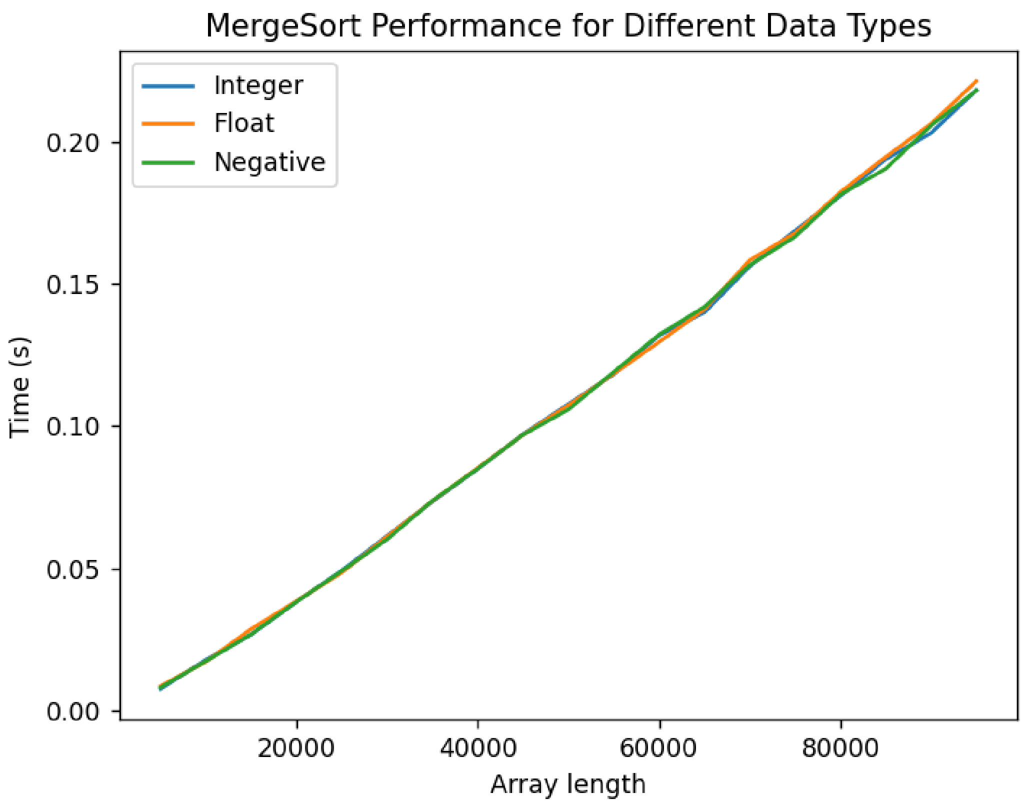


Figure 11. MergeSort results for large arrays with different data types.

As observed in the above images, MergeSort exhibits slower sorting times compared to QuickSort in all scenarios. Notably, for an array of one million integers, QuickSort outperforms MergeSort by 0.45 seconds. The graphical representation highlights the stability of MergeSort, sorting integers, negatives, and floats with nearly identical execution times and minimal variations compared to QuickSort. However, even with this stability, MergeSort is slower, surpassing 2 seconds for 100,000 elements, while QuickSort remains around 0.175 seconds.

Time Complexity:

T(N log(N)), Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation: $T(n) = 2T(n/2) + \theta(n)$. The above recurrence can be solved either using the Recurrence Tree method or the Master method. It falls in case II of the Master Method and the solution of the recurrence is $\theta(N\log(N))$. The time complexity of Merge Sort is $\theta(N\log(N))$ in all 3 cases (worst, average, and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

Auxiliary Space: O(N), In merge sort all elements are copied into an auxiliary array. So N auxiliary space is required for merge sort.

Applications of Merge Sort:

Sorting large datasets: Merge sort is particularly well-suited for sorting large datasets due to its guaranteed worst-case time complexity of O(n log n).

External sorting: Merge sort is commonly used in external sorting, where the data to be sorted is too large to fit into memory.

Custom sorting: Merge sort can be adapted to handle different input distributions, such as partially sorted, nearly sorted, or completely unsorted data.

Advantages of Merge Sort:

- Stability: Merge sort is a stable sorting algorithm, which means it maintains 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 logN), which means it performs well even on large datasets.
- Parallelizable: Merge sort is a naturally parallelizable algorithm, which means it can be easily parallelized to take advantage of multiple processors or threads.

Drawbacks of Merge Sort:

- Space complexity: Merge sort requires additional memory to store the merged sub-arrays during the sorting process.
- Not in-place: Merge sort is not an in-place sorting algorithm, which means it requires additional memory to store the sorted data. This can be a disadvantage in applications where memory usage is a concern.
- Not always optimal for small datasets: For small datasets, Merge sort has a higher time complexity than some other sorting algorithms, such as insertion sort. This can result in slower performance for very small datasets.

Heap Sort:

Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to the selection sort where we first find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements.

Algorithm Description:

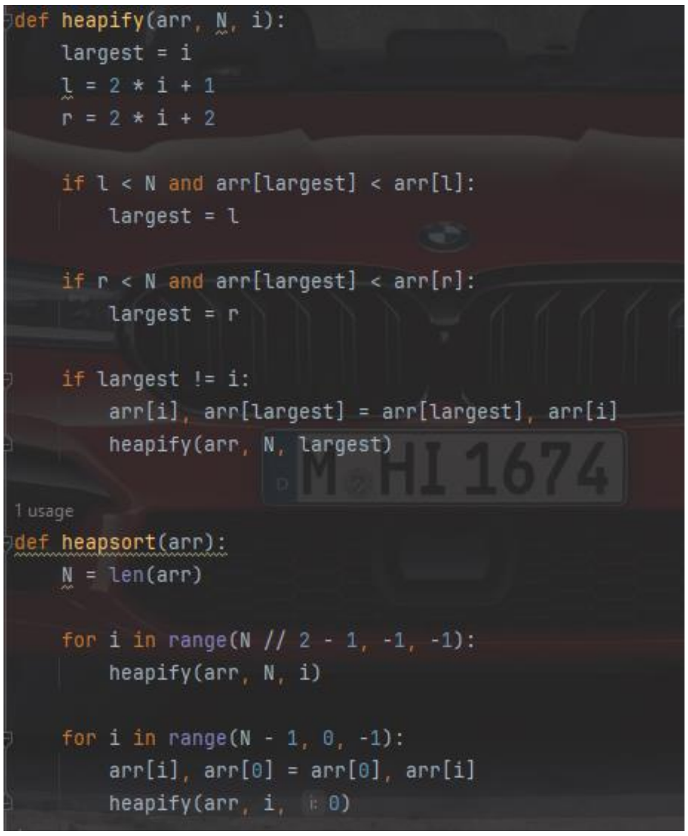
First convert the array into heap data structure using heapify, then one by one delete the root node of the Max-heap and replace it with the last node in the heap and then heapify the root of the heap. Repeat this process until size of heap is greater than 1.

- 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.

The naive recursive HeapSort method follows the algorithm as shown in the next pseudocode:

```
heapify(arr, N, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if l < N and arr[largest] < arr[l]:
        largest = l
    if r < N and arr[largest] < arr[r]:
        largest = r
    if largest ≠ i:
        swap(arr[i], arr[largest])
        heapify(arr, N, largest)

heapsort(arr):
    N = length of arr
    // Build a max heap
    for i from N/2 - 1 to 0 step -1:
        heapify(arr, N, i)
    // Extract elements from the heap
    for i from N-1 to 1 step -1:
        swap(arr[i], arr[0])
        heapify(arr, i, 0)
```

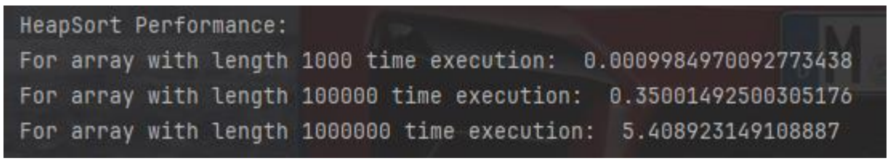
Implementation:



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

    if l < N and arr[largest] < 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)

1 usage
def heapsort(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)
```

Figure 12. HeapSort in Python.

Results:



```
HeapSort Performance:
For array with length 1000 time execution:  0.0009984970092773438
For array with length 100000 time execution:  0.35001492500305176
For array with length 1000000 time execution:  5.408923149108887
```

Figure 13. HeapSort results for large arrays with integer numbers.

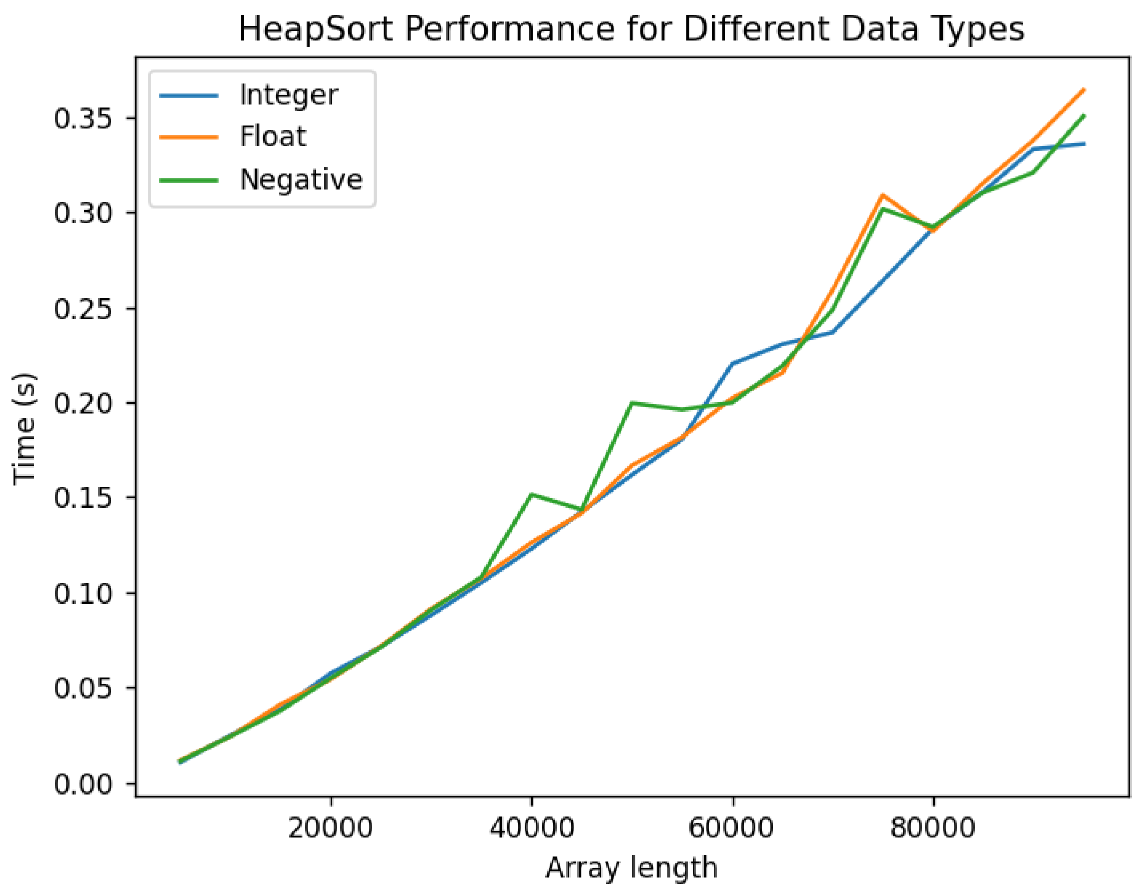


Figure 14. HeapSort results for large arrays with different data types.

As evident from the preceding visuals, HeapSort proves to be slower than MergeSort and QuickSort. It exhibited superior performance only when sorting an array of positive integers with a length of 1000, surpassing both MergeSort and QuickSort in speed. However, as the array length increased, HeapSort's efficiency diminished significantly. For a million positive integers, it lagged behind MergeSort by 2.55 seconds. Upon closer examination of the graph depicting the sorting times for arrays of 100k elements across integer, float, and negative data types, HeapSort consistently took around 0.35 seconds. In contrast, MergeSort hovered around 0.2 seconds, and QuickSort even less. Notably, around arrays of 50k elements, there was a considerable deviation for negative integers, which took longer to sort compared to the other types. Nevertheless, towards the end, all three types converged to a similar sorting time, with a minor deviation of a few milliseconds.

Time Complexity:

O(N log N)

Auxiliary Space: O(log n), due to the recursive call stack. However, auxiliary space can be O(1) for iterative implementation.

Important points about Heap Sort:

- Heap sort is an in-place algorithm.
- Its typical implementation is not stable but can be made stable
- Typically 2-3 times slower than well-implemented QuickSort. The reason for slowness is a lack of locality of reference.

Advantages of Heap Sort:

- Efficient Time Complexity: Heap Sort has a time complexity of O(n log n) in all cases. This makes it efficient for sorting large datasets. The log n factor comes from the height of the binary heap, and it ensures that the algorithm maintains good performance even with a large number of elements.
- Memory Usage: Memory usage can be minimal because apart from what is necessary to hold the initial list of items to be sorted, it needs no additional memory space to work
- Simplicity: It is simpler to understand than other equally efficient sorting algorithms because it does not use advanced computer science concepts such as recursion.

Disadvantages of Heap Sort:

- Costly: Heap sort is costly.
- Unstable: Heap sort is unstable. It might rearrange the relative order.
- Efficient: Heap Sort is not very efficient when working with highly complex data.

Bubble Sort:

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:

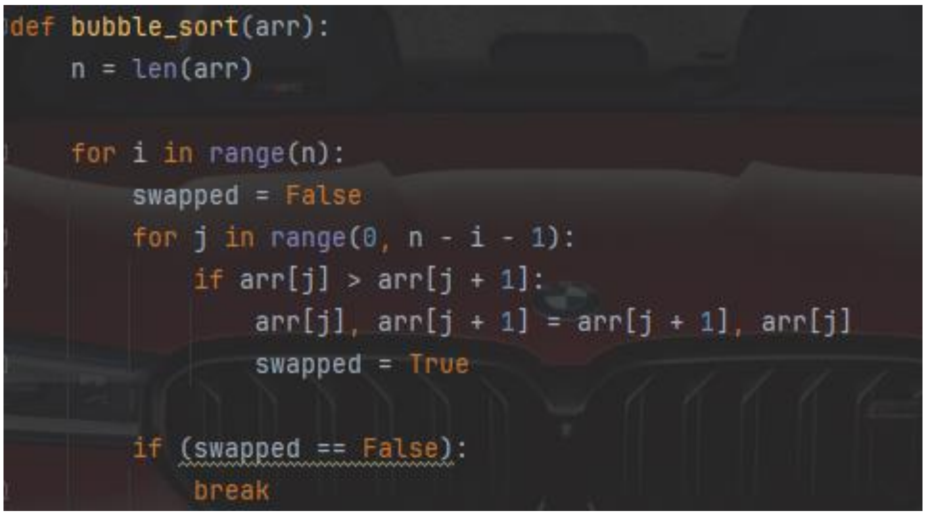
In Bubble Sort algorithm,

- Traverse from left and compare adjacent elements and the higher one is placed at right side.
- In this way, the largest element is moved to the rightmost end at first.
- This process is then continued to find the second largest and place it and so on until the data is sorted.

The naive recursive BubbleSort method follows the algorithm as shown in the next pseudocode:

```
bubble_sort(arr):
    n = length of arr

    // Iterate through the array
    for i from 0 to n-1:
        swapped = False

        // Iterate through the unsorted part of the array
        for j from 0 to n-i-1:
            // If adjacent elements are out of order, swap them
            if arr[j] > arr[j + 1]:
                swap(arr[j], arr[j + 1])
                swapped = True

        // If no two elements were swapped, array is already sorted
        if (swapped == False):
            break
```

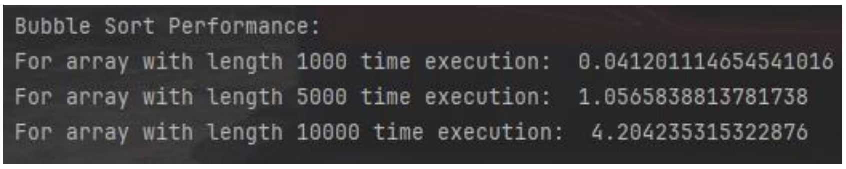
Implementation:



```python
def bubble_sort(arr):
    n = len(arr)

    for i in range(n):
        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 (swapped == False):
            break
```

Figure 15. Bubble Sort in Python.

Results:



```
Bubble Sort Performance:
For array with length 1000 time execution:  0.041201114654541016
For array with length 5000 time execution:  1.0565838813781738
For array with length 10000 time execution:  4.204235315322876
```

Figure 16. Bubble Sort results for large arrays with integer numbers.

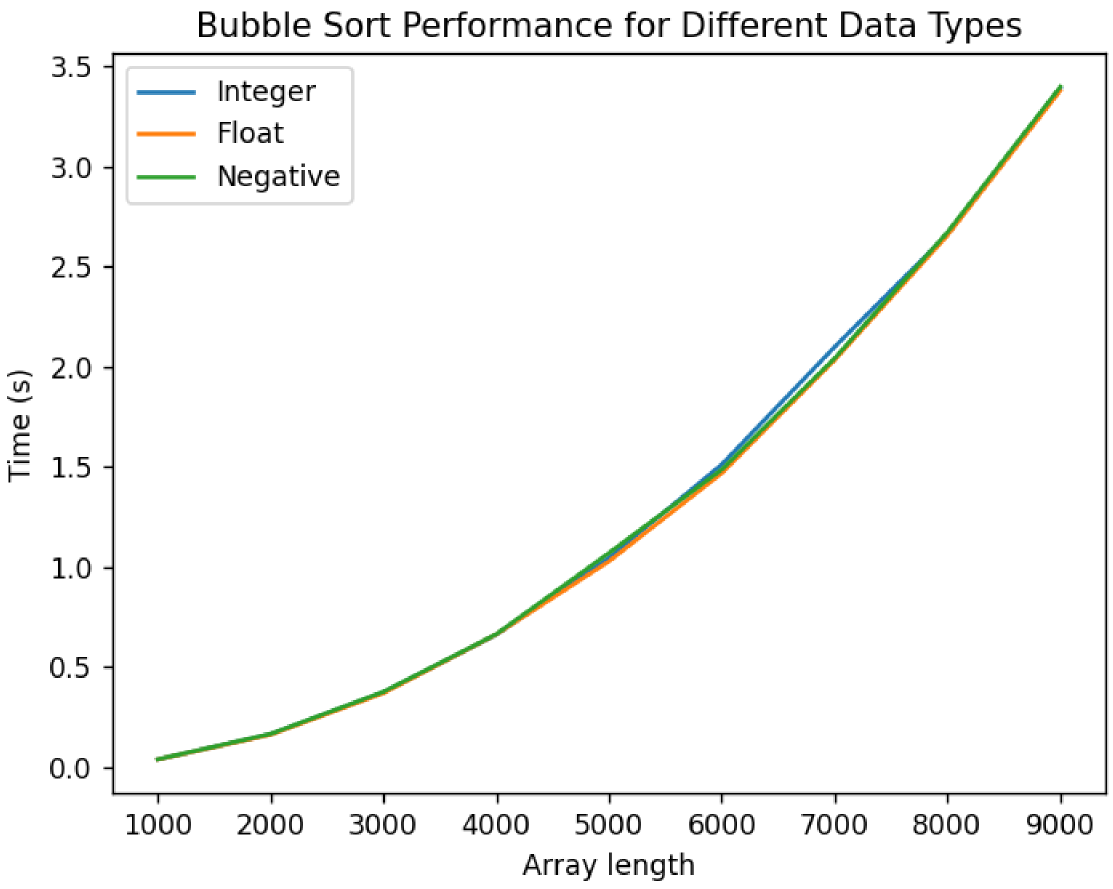


Figure 17. Bubble Sort results for large arrays with different data types.

In the case of Bubble Sort, a limited range was chosen for the lengths of the arrays due to its high time complexity of N^2, resulting in a significant increase in execution time. Across all scenarios, Bubble Sort consistently performed much slower, attributed to the presence of nested for loops, contributing to a quadratic time complexity. As depicted in the graph, the variation in execution time among different data types—integer, float, and negative—was nearly negligible, especially for arrays of up to 10k in length. This underscores Bubble Sort's inefficiency for larger datasets due to its quadratic nature.

Time Complexity: O(N2)
Auxiliary Space: O(1)

Advantages of Bubble Sort:

- Bubble sort is easy to understand and implement.
- It does not require any additional memory space.
- It is a stable sorting algorithm, meaning that elements with the same key value maintain their relative order in the sorted output.

Disadvantages of Bubble Sort:

- Bubble sort has a time complexity of O(N2) which makes it very slow for large data sets.
- Bubble sort is a comparison-based sorting algorithm, which means that it requires a comparison operator to determine the relative order of elements in the input data set. It can limit the efficiency of the algorithm in certain cases.

CONCLUSION

In conclusion, In all conducted scenarios, QuickSort emerged as the fastest sorting algorithm, showcasing remarkable efficiency in sorting a million-element integer array in just 2.4 seconds. Bubble Sort consistently proved to be the slowest due to its quadratic time complexity (O(N^2)). HeapSort exhibited superior performance only for small arrays with a length of 1000, surpassing both MergeSort and Bubble Sort in speed. HeapSort, despite its initial speed, lagged behind as the array length increased, emphasizing its inefficiency for larger datasets. MergeSort, while stable and exhibiting a consistent time complexity of O(N log N), fell behind QuickSort in terms of speed, surpassing 2 seconds for 100,000 elements. QuickSort's advantages include its efficiency on large datasets, with an average-case time complexity of O(N log N), making it one of the fastest sorting algorithms. In contrast, its disadvantages include the potential for a worst-case time complexity of O(N^2) when poorly chosen pivots, making it less suitable for small datasets. In summary, QuickSort's versatility and effectiveness make it a preferred choice for a wide range of scenarios, especially when dealing with large datasets. HeapSort, despite its initial speed, is less optimal for larger arrays, while MergeSort remains stable but slower compared to QuickSort. Bubble Sort, with its quadratic time complexity, is consistently the least efficient. Additionally, QuickSort's low overhead and efficient divide-and-conquer strategy further contribute to its superiority in various scenarios. While MergeSort's stability and consistent time complexity make it a reliable choice, it tends to exhibit slower performance compared to QuickSort, particularly for larger datasets.