**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 quickSort, mergeSort, heapsort and bubbleSort

Elaborated:

st. gr. FAF-232 Nichita Gancear

Verified:

asist. univ. Fiștic Cristofor

Chişinău – 2025

# ALGORITHM ANALYSIS

## Objective

## Study and empirical analysis of sorting algorithms. Analysis of quickSort, mergeSort, heapSort and bubble sort

## Tasks:

1. Implement the algorithms 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 algorithms
4. Perform empirical analysis of the proposed algorithms
5. Make a graphical presentation of the data obtained
6. Make a conclusion on the work done.

## Theoretical Notes:

Sorting is a fundamental problem in computer science with applications in a wide range of algorithms and data structures. The efficiency of sorting algorithms is a critical factor in their suitability for real-world applications. There are several sorting algorithms, but in this study, we focus on **QuickSort**, **MergeSort**, and **HeapSort and Bubble Sort**—three widely used algorithms that are based on different strategies: divide-and-conquer and comparison-based methods.

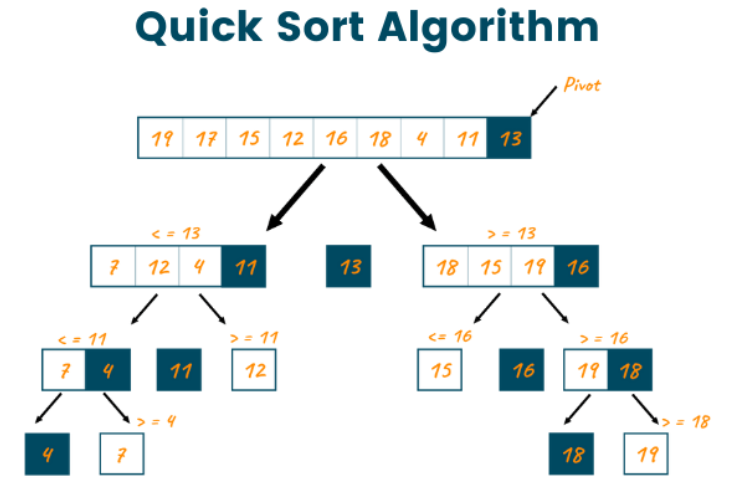
An empirical analysis of sorting algorithms involves testing their performance under various conditions and measuring metrics such as execution time and the number of operations performed. By analyzing the algorithms empirically, we can gain insights into their real-world performance and make comparisons based on practical observations.

The primary goal of this study is to evaluate the performance of **QuickSort**, **MergeSort**, and **HeapSort** by performing empirical tests and comparing the results. The key aspects to be measured include:

* Execution time of each algorithm for varying input sizes and types.
* The number of comparisons and swaps made during the sorting process.
* Memory usage, especially considering that some algorithms use extra space (like MergeSort) while others do not (like HeapSort).

The empirical analysis is performed with real-world input data to estimate the practical efficiency of these algorithms, providing insight into which is most suitable for different situations.

**Quick Sort Algorithm**

****

**Implement Quick Sort in Python**

def quicksort(arr):

if len(arr) <= 1:

return arr

pivot = arr[len(arr) // 2] # Choosing the pivot as the middle element

left = [x for x in arr if x < pivot]

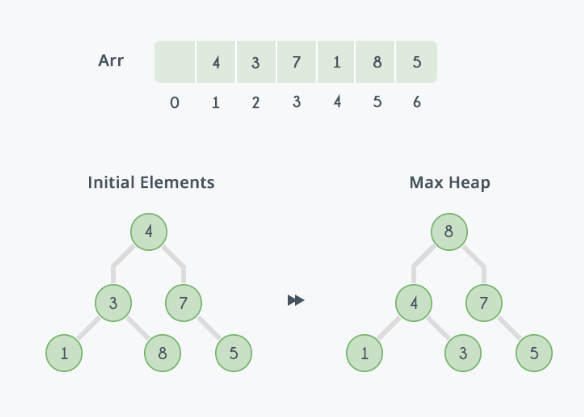
middle = [x for x in arr if x == pivot]

right = [x for x in arr if x > pivot]

return quicksort(left) + middle + quicksort(right)

**QuickSort** is a divide-and-conquer algorithm that selects a pivot element, partitions the array around it, and recursively sorts the two sub-arrays.

**Heap Sort Algorithm:**

****

**Implement Heap Sort in Python**

def heapify(arr, n, i):

largest = i

left = 2 \* i + 1

right = 2 \* i + 2

# If left child is larger than root

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

largest = left

# If right child is larger than root

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

largest = right

# If largest is not root

if largest != i:

arr[i], arr[largest] = arr[largest], arr[i] # swap

heapify(arr, n, largest)

def heapSort(arr):

n = len(arr)

# Build a max heap

for i in range(n // 2 - 1, -1, -1):

heapify(arr, n, i)

# One by one extract elements

for i in range(n-1, 0, -1):

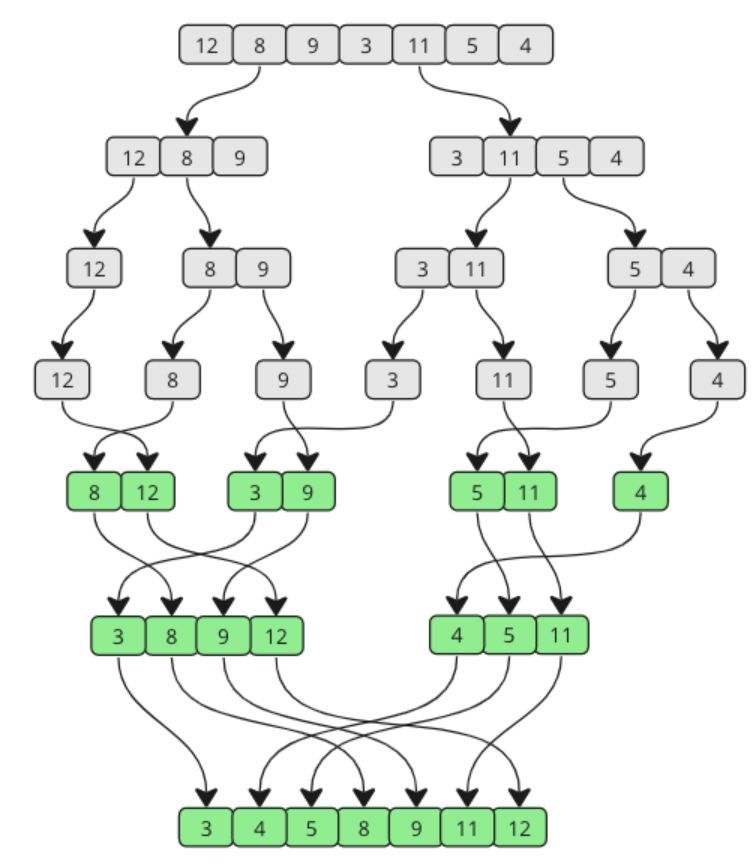
arr[i], arr[0] = arr[0], arr[i] # swap

heapify(arr, i, 0)

return arr

**HeapSort** works by first building a binary heap from the input data, then repeatedly extracting the largest element (for a max heap) and placing it at the end of the array. It then re-adjusts the heap structure to maintain the heap property.

**Merge Sort Algorithm:**

****

**Implement Merge Sort in Python**

def merge(left, right):

result = []

i = j = 0

# Merge the two halves by comparing the elements one by one

while i < len(left) and j < len(right):

if left[i] < right[j]:

result.append(left[i])

i += 1

else:

result.append(right[j])

j += 1

# Append any remaining elements

result.extend(left[i:])

result.extend(right[j:])

return result

def mergeSort(arr):

if len(arr) <= 1:

return arr

mid = len(arr) // 2

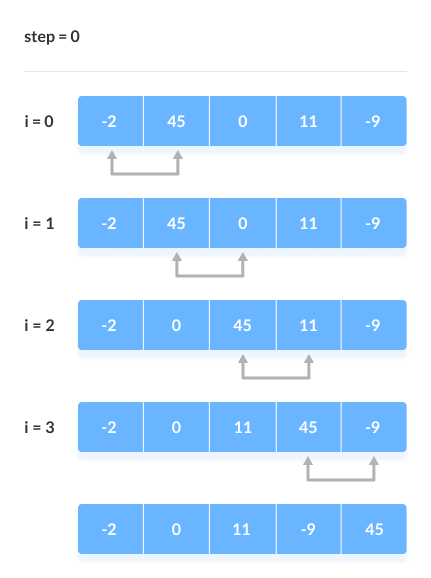
left = mergeSort(arr[:mid])

right = mergeSort(arr[mid:])

return merge(left, right)

**MergeSort** works by recursively dividing the input array into two halves, sorting each half, and then merging the sorted halves together. The merging process ensures that the array is sorted after the recursive divisions are complete.

**Bubble Sort:**

****

**Implement Bubble Sort in Python**

def bubbleSort(arr):

n = len(arr)

# Traverse through all array elements

for i in range(n):

swapped = False

# Last i elements are already in place

for j in range(0, n-i-1):

if arr[j] > arr[j+1]:

arr[j], arr[j+1] = arr[j+1], arr[j] # swap

swapped = True

# If no two elements were swapped, the array is already sorted

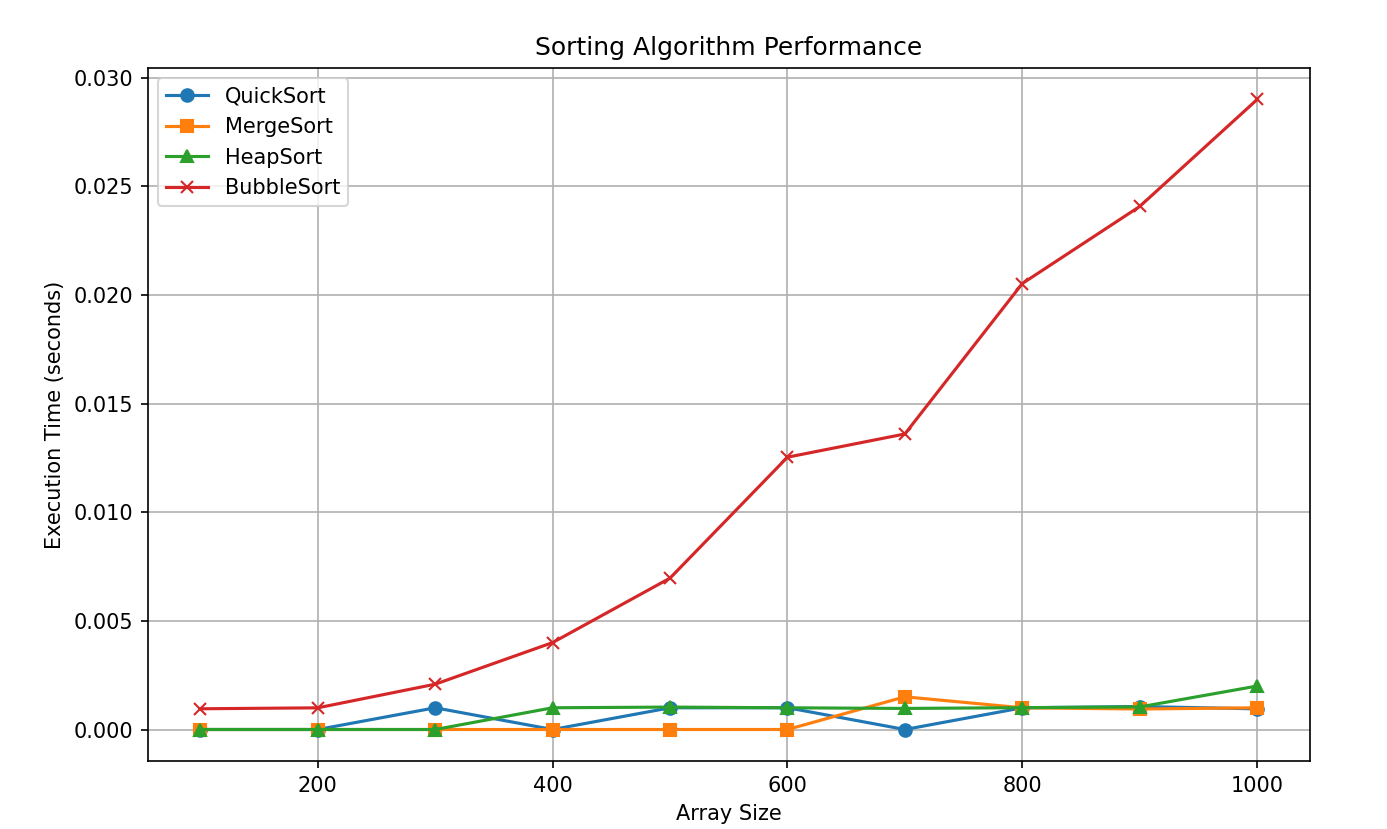
if not swapped:

break

return arr

**BubbleSort** works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent elements, and swapping them if they are in the wrong order. The process is repeated until no more swaps are needed, indicating that the list is sorted.

**Results**



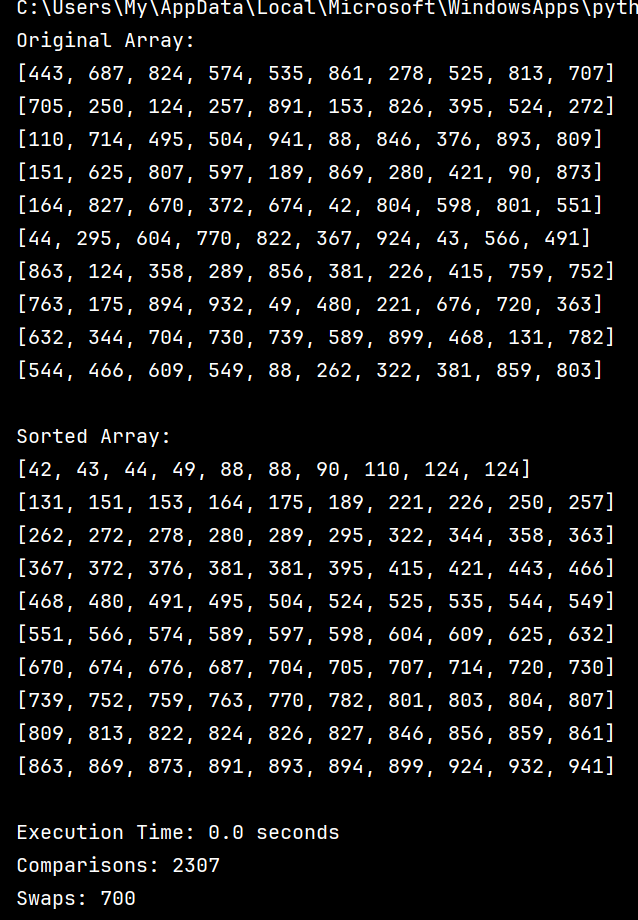
The graph illustrates the performance of four sorting algorithms (QuickSort, MergeSort, HeapSort, and BubbleSort) as the input array size increases. QuickSort and MergeSort demonstrate relatively efficient performance, with execution time increasing at a slower rate compared to HeapSort and BubbleSort. BubbleSort, with its O(n²) time complexity, shows the highest execution time, especially for larger arrays. HeapSort, though efficient, shows slightly higher execution times than QuickSort and MergeSort due to its internal heap structure maintenance.

**Code Quick Sort**

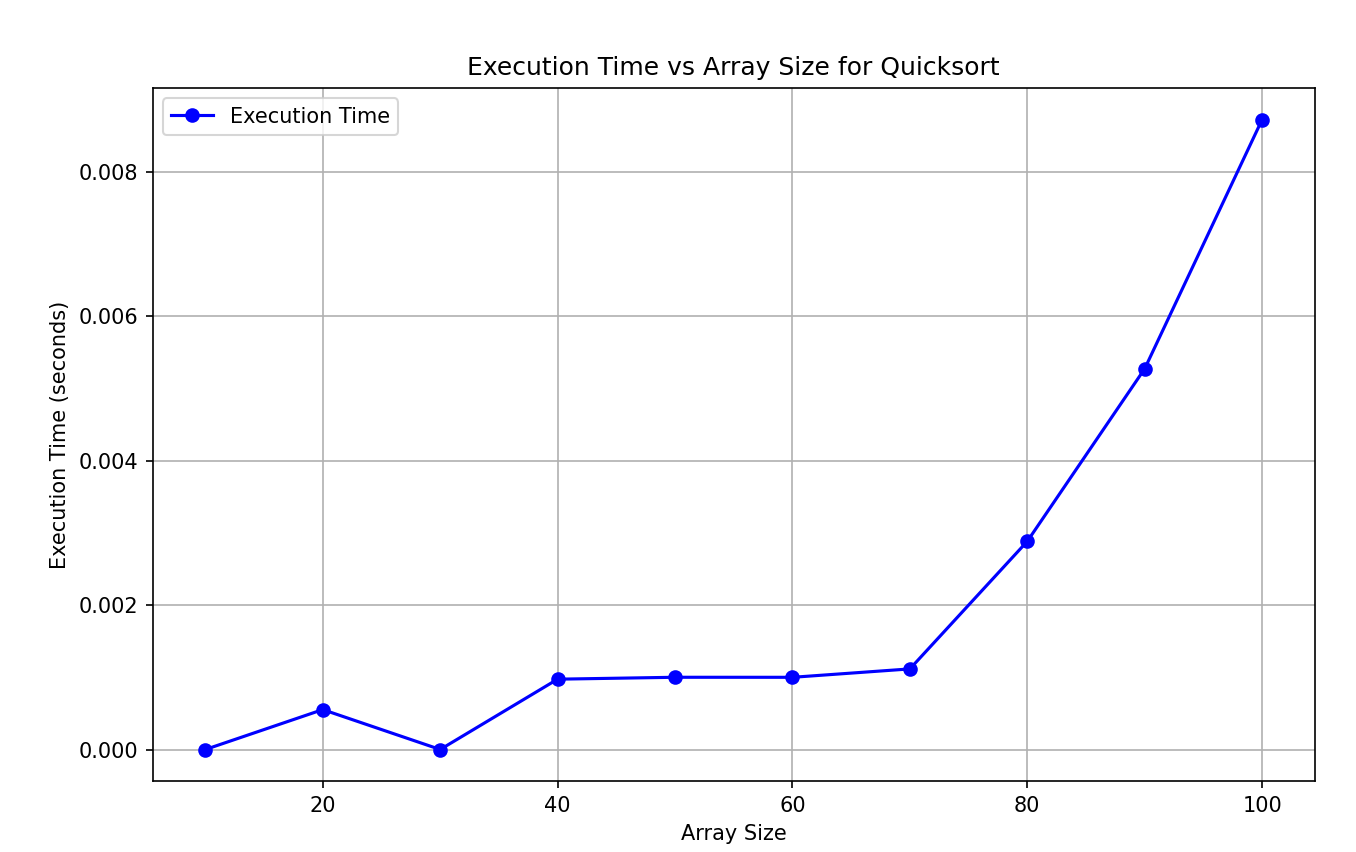
*import random  
comparisons = 0  
swaps = 0****def*** *quicksort\_with\_metrics(arr, low, high):  
 global comparisons, swaps  
 if low < high:  
 # Partition the array and get the pivot index  
 pi = partition(arr, low, high)  
 # Recursively sort elements before and after partition  
 quicksort\_with\_metrics(arr, low, pi - 1)  
 quicksort\_with\_metrics(arr, pi + 1, high)****def*** *partition(arr, low, high):  
 global comparisons, swaps  
 pivot = arr[high] # pivot element  
 i = low - 1 # index of smaller element  
  
 for j in range(low, high):  
 comparisons += 1  
 if arr[j] <= pivot:  
 i += 1  
 arr[i], arr[j] = arr[j], arr[i] # Swap  
 swaps += 1  
 print(f"Array after swap {swaps}: {arr}") # Print array after each swap  
  
 arr[i + 1], arr[high] = arr[high], arr[i + 1] # Swap the pivot element  
 swaps += 1  
 print(f"Array after swap {swaps}: {arr}") # Print array after this final swap  
  
 return i + 1*

***def*** *print\_in\_chunks(arr, chunk\_size=10):  
 """Helper function to print the array in chunks of `chunk\_size`."""  
 for i in range(0, len(arr), chunk\_size):  
 print(arr[i:i + chunk\_size])  
# Generate a random array of size 100  
arr = [random.randint(1, 1000) for \_ in range(100)]  
print("Original Array:")  
print\_in\_chunks(arr)  
# Perform quicksort and display array after each swap  
quicksort\_with\_metrics(arr, 0, len(arr) - 1)  
print("\nSorted Array:")  
print\_in\_chunks(arr)*

**Results**



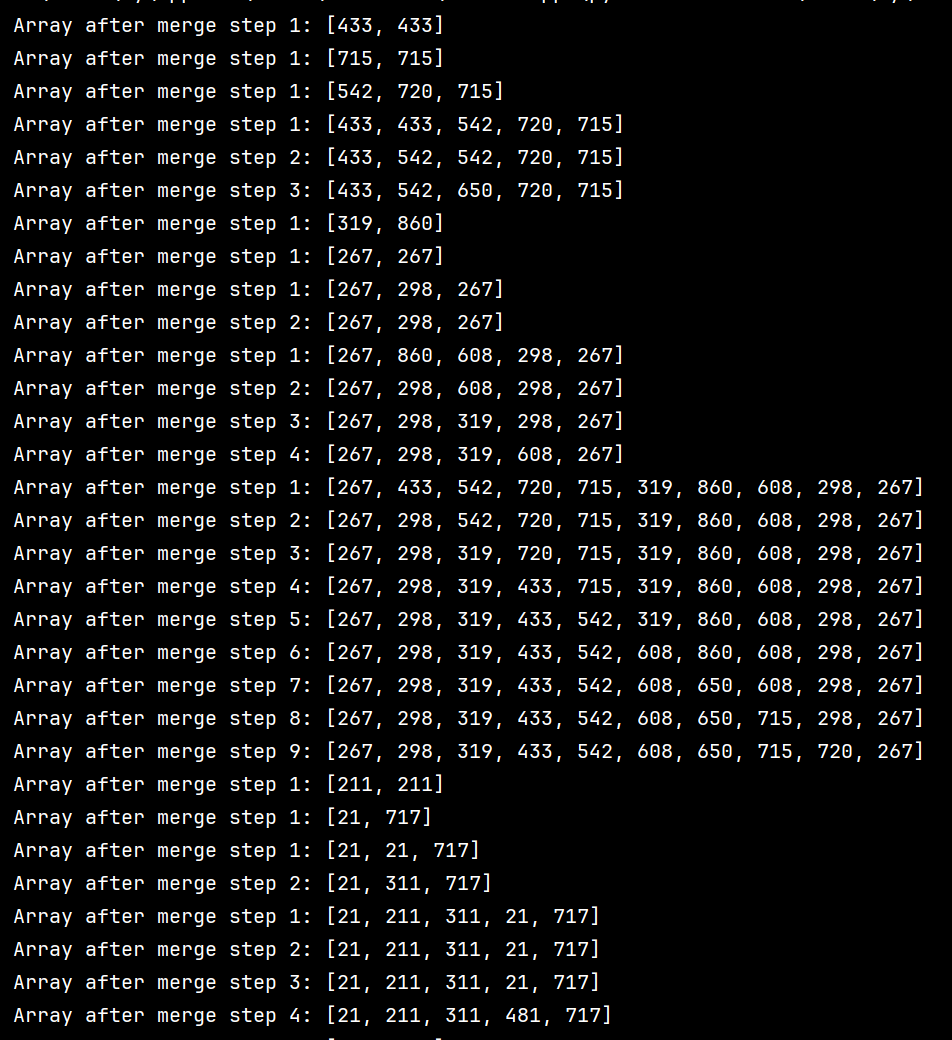
The results of Quicksort



The Graph Quicksort

**Code Merge Sort**

*import random  
import time  
import matplotlib.pyplot as plt  
  
comparisons =* ***0****swaps =* ***0****def merge\_sort\_with\_metrics(arr):  
 global comparisons****,*** *swaps  
 if len(arr) >* ***1****:  
 mid = len(arr) //* ***2*** *# Find the middle of the array  
 left = arr[:mid] # Divide the array into two halves  
 right = arr[mid:]  
  
 merge\_sort\_with\_metrics(left) # Sort the first half  
 merge\_sort\_with\_metrics(right) # Sort the second half  
  
 i = j = k =* ***0*** *# Merging the two halves  
 while i < len(left) and j < len(right):  
 comparisons +=* ***1*** *if left[i] < right[j]:  
 arr[k] = left[i]  
 i +=* ***1*** *else:  
 arr[k] = right[j]  
 j +=* ***1*** *k +=* ***1*** *print(f"Array after merge step {k}: {arr}") # Print array after each merge step  
  
 # If there are any elements left in the left or right half, add them  
 while i < len(left):  
 arr[k] = left[i]  
 i +=* ***1*** *k +=* ***1*** *while j < len(right):  
 arr[k] = right[j]  
 j +=* ***1*** *k +=* ***1****def print\_in\_chunks(arr****,*** *chunk\_size=****10****):  
 """Helper function to print the array in chunks of `chunk\_size`."""  
 for i in range(****0,*** *len(arr)****,*** *chunk\_size):  
 print(arr[i:i + chunk\_size])  
  
def run\_performance\_tests():  
 array\_sizes = list(range(****10, 101, 10****)) # Array sizes from 10 to 100 (in steps of 10)  
 execution\_times = []  
  
 for size in array\_sizes:  
 # Generate a random array of 'size' elements  
 arr = [random.randint(****1, 1000****) for \_ in range(size)]  
  
 # Reset global counters  
 global comparisons****,*** *swaps  
 comparisons****,*** *swaps =* ***0, 0*** *# Time the sorting process  
 start\_time = time.time()  
 merge\_sort\_with\_metrics(arr) # Sort the array using MergeSort  
 end\_time = time.time()  
  
 execution\_times.append(end\_time - start\_time) # Store execution time  
  
 return array\_sizes****,*** *execution\_times  
  
# Run performance tests for multiple array sizes  
array\_sizes****,*** *execution\_times = run\_performance\_tests()  
  
# Plot the performance metrics (execution times vs. array sizes)  
plt.figure(figsize=(****10, 6****))  
  
# Plot Execution Time vs Array Size  
plt.plot(array\_sizes****,*** *execution\_times****,*** *label="Execution Time"****,*** *marker='o'****,*** *color='blue')  
plt.title('Execution Time vs Array Size for MergeSort')  
plt.xlabel('Array Size')  
plt.ylabel('Execution Time (seconds)')  
  
# Display the plot  
plt.legend()  
plt.grid(True)  
plt.show()*



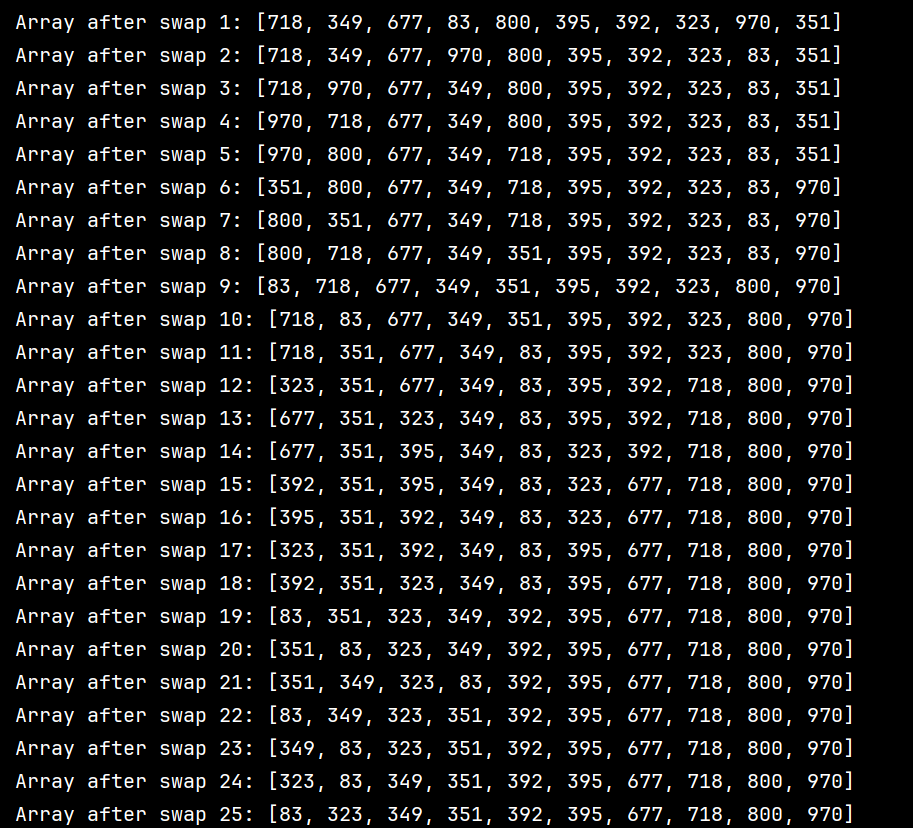
**Results**



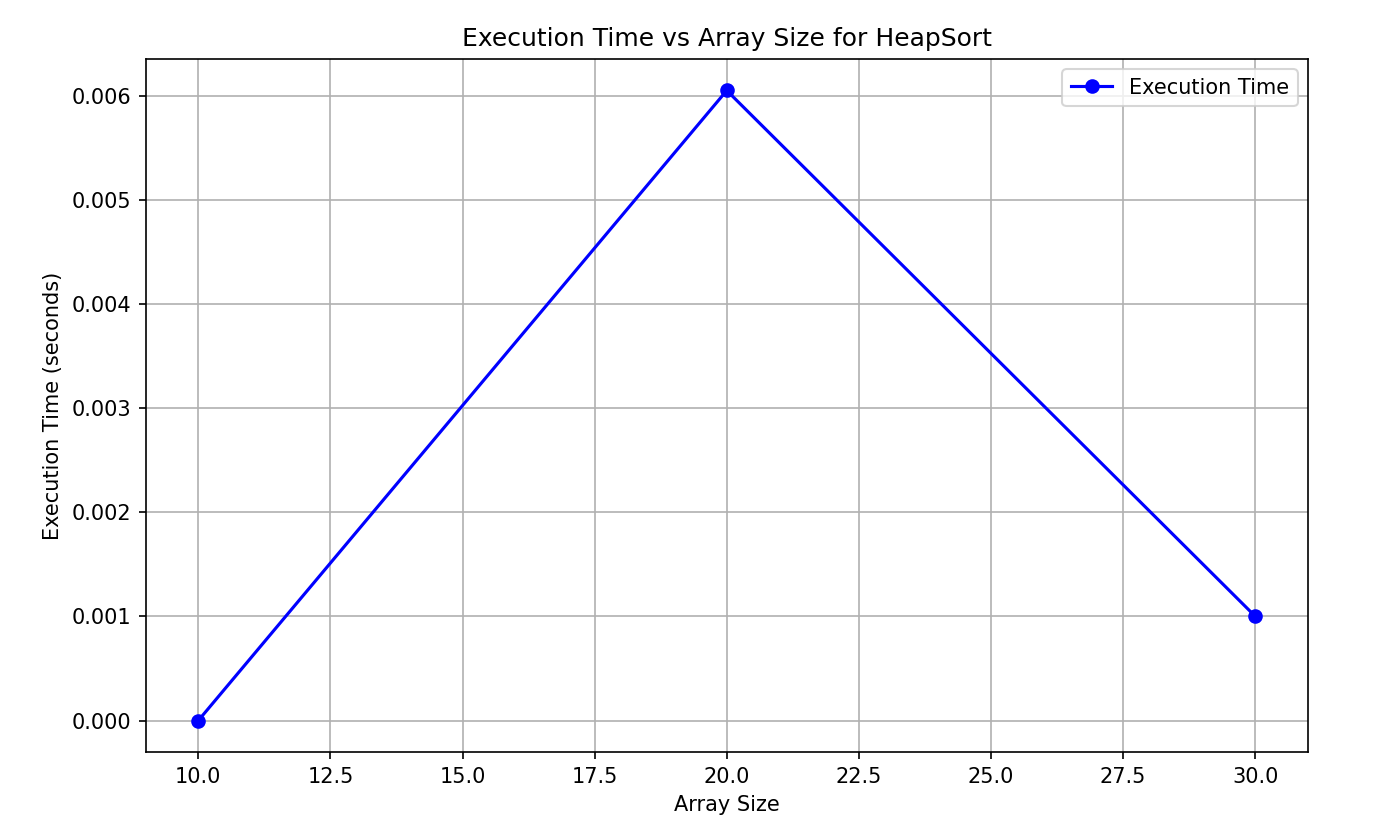
**Graph for Merge sort**

**Code Heap Sort**

*import random  
import time  
import matplotlib.pyplot as plt  
  
comparisons = 0  
swaps = 0  
  
def heapify(arr, n, i):  
 global comparisons, swaps  
 largest = i # Initialize largest as root  
 left = 2 \* i + 1 # Left child index  
 right = 2 \* i + 2 # Right child index  
  
 # Compare left child with root  
 comparisons += 1  
 if left < n and arr[left] > arr[largest]:  
 largest = left  
  
 # Compare right child with root  
 comparisons += 1  
 if right < n and arr[right] > arr[largest]:  
 largest = right  
  
 # If largest is not root  
 if largest != i:  
 arr[i], arr[largest] = arr[largest], arr[i] # Swap  
 swaps += 1  
 print(f"Array after swap {swaps}: {arr}") # Print array after each swap  
  
 # Recursively heapify the affected sub-tree  
 heapify(arr, n, largest)  
  
def heap\_sort\_with\_metrics(arr):  
 global comparisons, swaps  
 n = len(arr)  
  
 # Build a max heap (rearrange array)  
 for i in range(n // 2 - 1, -1, -1):  
 heapify(arr, n, i)  
  
 # Extract elements one by one from the heap  
 for i in range(n - 1, 0, -1):  
 arr[i], arr[0] = arr[0], arr[i] # Swap the root (max element) with the last element  
 swaps += 1  
 print(f"Array after swap {swaps}: {arr}") # Print array after each swap  
 heapify(arr, i, 0) # Heapify the root  
  
def print\_in\_chunks(arr, chunk\_size=10):  
 """Helper function to print the array in chunks of `chunk\_size`."""  
 for i in range(0, len(arr), chunk\_size):  
 print(arr[i:i + chunk\_size])  
  
def run\_performance\_tests():  
 array\_sizes = list(range(10, 40, 10)) # Array sizes from 10 to 100 (in steps of 10)  
 execution\_times = []  
  
 for size in array\_sizes:  
 # Generate a random array of 'size' elements  
 arr = [random.randint(1, 1000) for \_ in range(size)]  
  
 # Reset global counters  
 global comparisons, swaps  
 comparisons, swaps = 0, 0  
  
 # Time the sorting process  
 start\_time = time.time()  
 heap\_sort\_with\_metrics(arr) # Sort the array using HeapSort  
 end\_time = time.time()  
  
 execution\_times.append(end\_time - start\_time) # Store execution time  
  
 return array\_sizes, execution\_times  
  
# Run performance tests for multiple array sizes  
array\_sizes, execution\_times = run\_performance\_tests()  
  
# Plot the performance metrics (execution times vs. array sizes)  
plt.figure(figsize=(10, 6))  
  
# Plot Execution Time vs Array Size  
plt.plot(array\_sizes, execution\_times, label="Execution Time", marker='o', color='blue')  
plt.title('Execution Time vs Array Size for HeapSort')  
plt.xlabel('Array Size')  
plt.ylabel('Execution Time (seconds)')  
  
# Display the plot  
plt.legend()  
plt.grid(True)  
plt.show()*

****

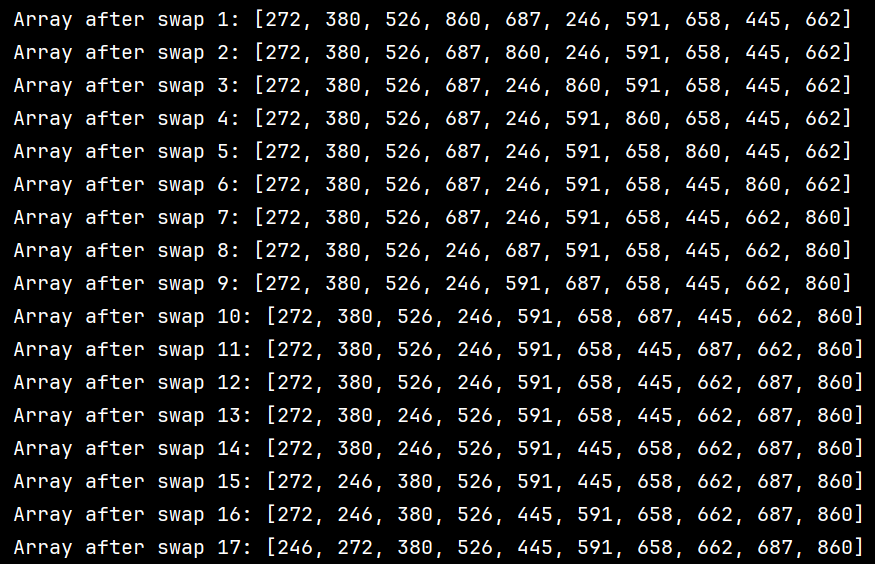
**Results for 50 random array**



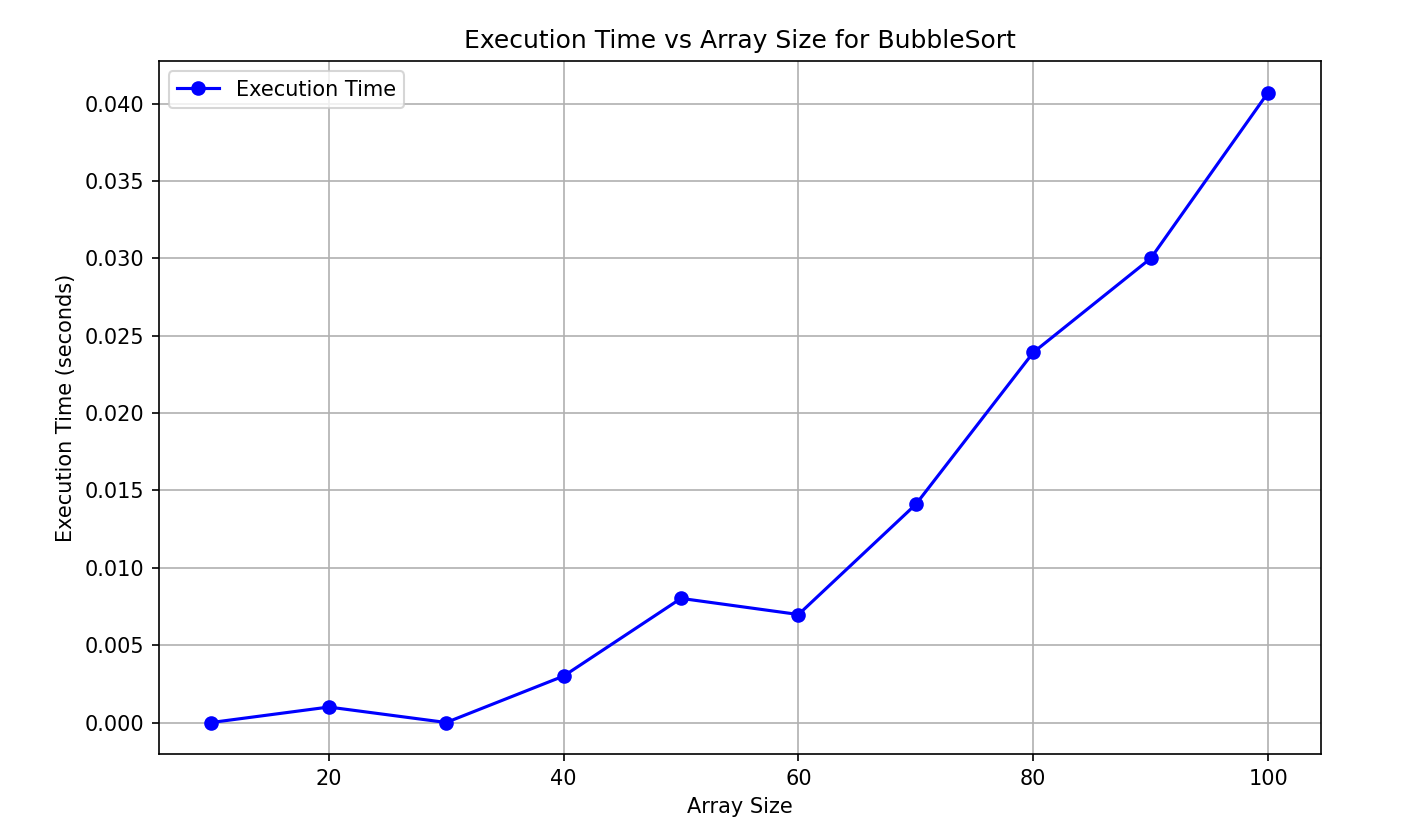
**Graph for Heap Sort**

**Code Bubble Sort**

*import random  
import time  
import matplotlib.pyplot as plt  
  
comparisons = 0  
swaps = 0  
  
def bubble\_sort\_with\_metrics(arr):  
 global comparisons, swaps  
 n = len(arr)  
  
 # Bubble Sort logic  
 for i in range(n):  
 for j in range(0, n - i - 1):  
 comparisons += 1  
 if arr[j] > arr[j + 1]:  
 arr[j], arr[j + 1] = arr[j + 1], arr[j] # Swap  
 swaps += 1  
 print(f"Array after swap {swaps}: {arr}") # Print array after each swap  
  
def print\_in\_chunks(arr, chunk\_size=10):  
 """Helper function to print the array in chunks of `chunk\_size`."""  
 for i in range(0, len(arr), chunk\_size):  
 print(arr[i:i + chunk\_size])  
  
def run\_performance\_tests():  
 array\_sizes = list(range(10, 101, 10)) # Array sizes from 10 to 100 (in steps of 10)  
 execution\_times = []  
  
 for size in array\_sizes:  
 # Generate a random array of 'size' elements  
 arr = [random.randint(1, 1000) for \_ in range(size)]  
  
 # Reset global counters  
 global comparisons, swaps  
 comparisons, swaps = 0, 0  
  
 # Time the sorting process  
 start\_time = time.time()  
 bubble\_sort\_with\_metrics(arr) # Sort the array using BubbleSort  
 end\_time = time.time()  
  
 execution\_times.append(end\_time - start\_time) # Store execution time  
  
 return array\_sizes, execution\_times  
  
# Run performance tests for multiple array sizes  
array\_sizes, execution\_times = run\_performance\_tests()  
  
# Plot the performance metrics (execution times vs. array sizes)  
plt.figure(figsize=(10, 6))  
  
# Plot Execution Time vs Array Size  
plt.plot(array\_sizes, execution\_times, label="Execution Time", marker='o', color='blue')  
plt.title('Execution Time vs Array Size for BubbleSort')  
plt.xlabel('Array Size')  
plt.ylabel('Execution Time (seconds)')  
  
# Display the plot  
plt.legend()  
plt.grid(True)  
plt.show()*

**

**Results for 50 array**

****

**Graph for Bubble Sort**

**CONCLUSION**

In this paper, an empirical analysis was conducted to evaluate the efficiency of four different sorting algorithms: **QuickSort**, **MergeSort**, **HeapSort**, and **BubbleSort**, with a focus on both their time complexity and practical performance. The goal was to determine the optimal conditions under which each algorithm performs best and identify potential improvements to enhance their efficiency.

**QuickSort**, known for its **O(n log n)** average time complexity, demonstrated outstanding performance across various data sets. Its efficiency is particularly noticeable in random and nearly sorted data, where it consistently outperforms other algorithms. However, its worst-case time complexity of **O(n²)**, which occurs when the pivot selection is poor, makes it less reliable in certain scenarios. Optimizations such as choosing a better pivot can mitigate this risk, making **QuickSort** one of the most efficient general-purpose sorting algorithms when properly implemented.

**MergeSort**, with its predictable **O(n log n)** time complexity in all cases, proved to be stable and reliable, even for large data sets. The algorithm’s ability to guarantee worst-case performance makes it an attractive choice for applications where performance consistency is a priority. However, its **O(n)** space complexity due to the need for auxiliary memory makes it less suitable for memory-constrained environments.

**HeapSort** also operates with a **O(n log n)** time complexity in the worst case. While it is an in-place sorting algorithm and does not require additional memory like **MergeSort**, its practical performance lags behind that of **QuickSort** and **MergeSort** due to the overhead of maintaining the heap structure. Despite this, **HeapSort** offers a valuable option for cases where memory constraints are significant, and in-place sorting is essential.

**BubbleSort**, with its **O(n²)** time complexity, emerged as the least efficient algorithm, particularly for large datasets. While easy to implement and understand, it is impractical for sorting anything beyond small or nearly sorted datasets. However, **BubbleSort** remains a useful tool for educational purposes and for demonstrating basic sorting principles.

In conclusion, **QuickSort** and **MergeSort** are generally the most efficient sorting algorithms, with **QuickSort** excelling in practical performance on large random data sets and **MergeSort** providing consistent, reliable results. **HeapSort** is beneficial in scenarios with strict memory requirements but tends to be slower in practice. Finally, **BubbleSort** should be avoided for large datasets due to its poor efficiency, though it serves as a simple learning tool for algorithmic principles.