MINISTRY OF EDUCATION, CULTURE AND RESEARCH OF REPUBLIC OF MOLDOVA

TECHNICAL UNIVERSITY OF MOLDOVA

FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS

DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS

Algorithm Analysis

Laboratory work 2 : Study and empirical analysis of sorting algorithms.

Elaborated:	
st.gr. FAF-211	Corețch Mihai
Verified:	
asist.univ.	Fiștic Cristofor

Chișinău, 2023

Content

Sorting algorithms are a fundamental concept in computer science that are used to rearrange collections of data in a specific order efficiently. There are various sorting algorithms, such as bubble sort, selection sort, insertion sort, merge sort, quicksort, and heapsort, each with its own approach and time and space complexity. Understanding sorting algorithms is essential for computer scientists to improve the performance of various applications such as databases, search engines, and operating systems.

Sorting algorithms can be internal or external, depending on the size of the data. Internal sorting algorithms are used for data that can fit into the main memory of a computer, while external sorting algorithms are used for data that is too large to fit into the main memory and must be sorted on external storage media. Sorting algorithms play a vital role in handling large data sets, and the efficiency of the algorithm can make a significant impact on performance.

Sorting algorithms have numerous real-world applications, from sorting names in a phone book to sorting large data sets in scientific research, and they are used in various industries, including finance, healthcare, and engineering. Efficient sorting algorithms are also used in combination with other algorithms to optimize complex operations, such as data compression and searching. Understanding sorting algorithms can help choose the most efficient algorithm for a given task, optimize code performance, and improve problem-solving skills.

QuickSort

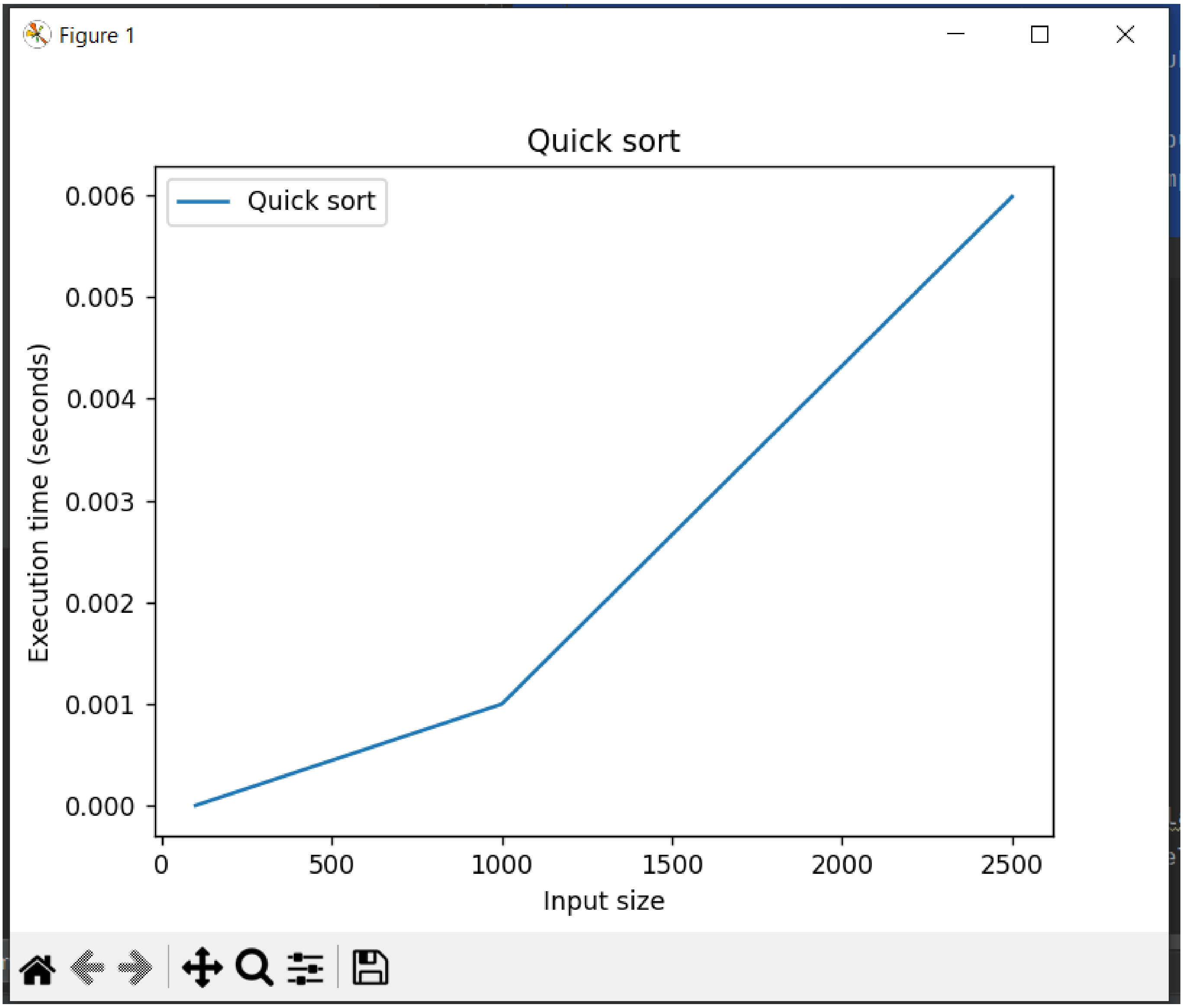
Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

- Always pick the first element as a pivot.
- Always pick the last element as a pivot (implemented below)
- Pick a random element as a pivot.
- Pick median as the pivot.

The key process in quickSort is a partition(). The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time. Quicksort 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. For this reason, it is sometimes called partition-exchange sort.[4] The sub-arrays are then sorted recursively. This can be done in-place, requiring small additional amounts of memory to perform the sorting.

Quicksort is a comparison sort, meaning that it can sort items of any type for which a "less-than" relation (formally, a total order) is defined. Most implementations of quicksort are not stable, meaning that the relative order of equal sort items is not preserved.

Mathematical analysis of quicksort shows that, on average, the algorithm takes O($n \log n$) comparisons to sort n items. In the worst case, it makes O(n^2) comparisons.



Code:

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

pivot = array_for_quick_sort[len(array_for_quick_sort) // 2]
left = [x for x in array_for_quick_sort if x < pivot]
middle = [x for x in array_for_quick_sort if x == pivot]
right = [x for x in array_for_quick_sort if x > pivot]

return quick_sort(left) + middle + quick_sort(right)
```

Mergesort

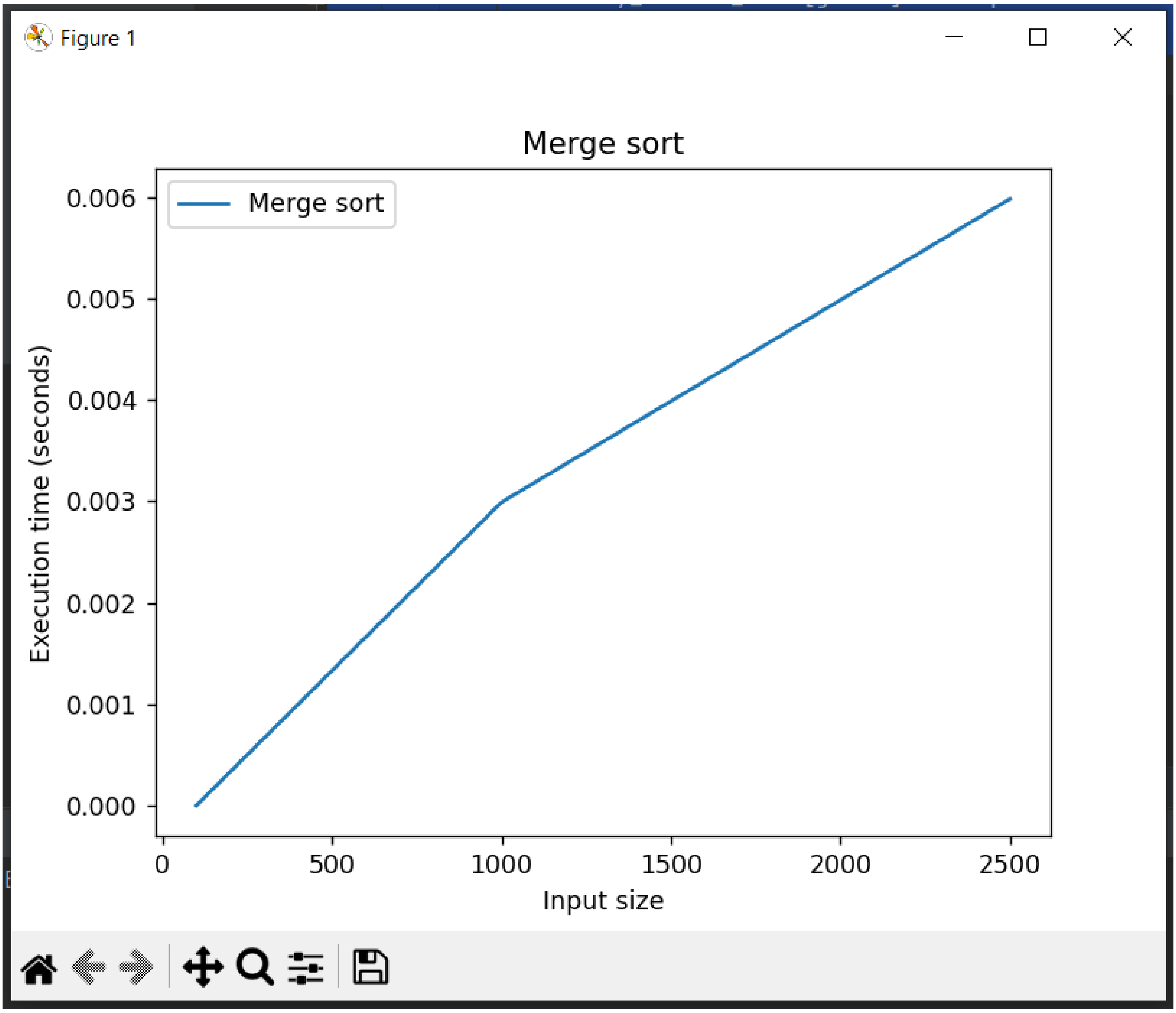
Merge sort is a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

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.

One thing that you might wonder is what is the specialty of this algorithm. We already have a number of sorting algorithms then why do we need this algorithm? One of the main advantages of merge sort is that it has a time complexity of O(n log n), which means it can sort large arrays relatively quickly. It is also a stable sort, which means that the order of elements with equal values is preserved during the sort.

Merge sort is a popular choice for sorting large datasets because it is relatively efficient and easy to implement. It is often used in conjunction with other algorithms, such as quicksort, to improve the overall performance of a sorting routine.

Think of it as a recursive algorithm continuously splits the array in half until it cannot be further divided. This means that if the array becomes empty or has only one element left, the dividing will stop, i.e. it is the base case to stop the recursion. If the array has multiple elements, split the array into halves and recursively invoke the merge sort on each of the halves. Finally, when both halves are sorted, the merge operation is applied. Merge operation is the process of taking two smaller sorted arrays and combining them to eventually make a larger one.



Code:

```
def merge_sort(array_for_merge_sort):
if len(array_for_merge_sort) <= 1:
    return array_for_merge_sort

# Split the array into two halves
mid = len(array_for_merge_sort) // 2
left = array_for_merge_sort[:mid]
right = array_for_merge_sort[mid:]

# Recursively sort each half
left_sorted = merge_sort(left)
right_sorted = merge_sort(right)
```

```
    # Merge the sorted halves
    i = j = 0
    merged = []
    while i < len(left_sorted) and j < len(right_sorted):
        if left_sorted[i] < right_sorted[j]:
            merged.append(left_sorted[i])
            i += 1
        else:
            merged.append(right_sorted[j])
            j += 1

    merged += left_sorted[i:]
    merged += right_sorted[j:]

    return merged
```

HeapSort

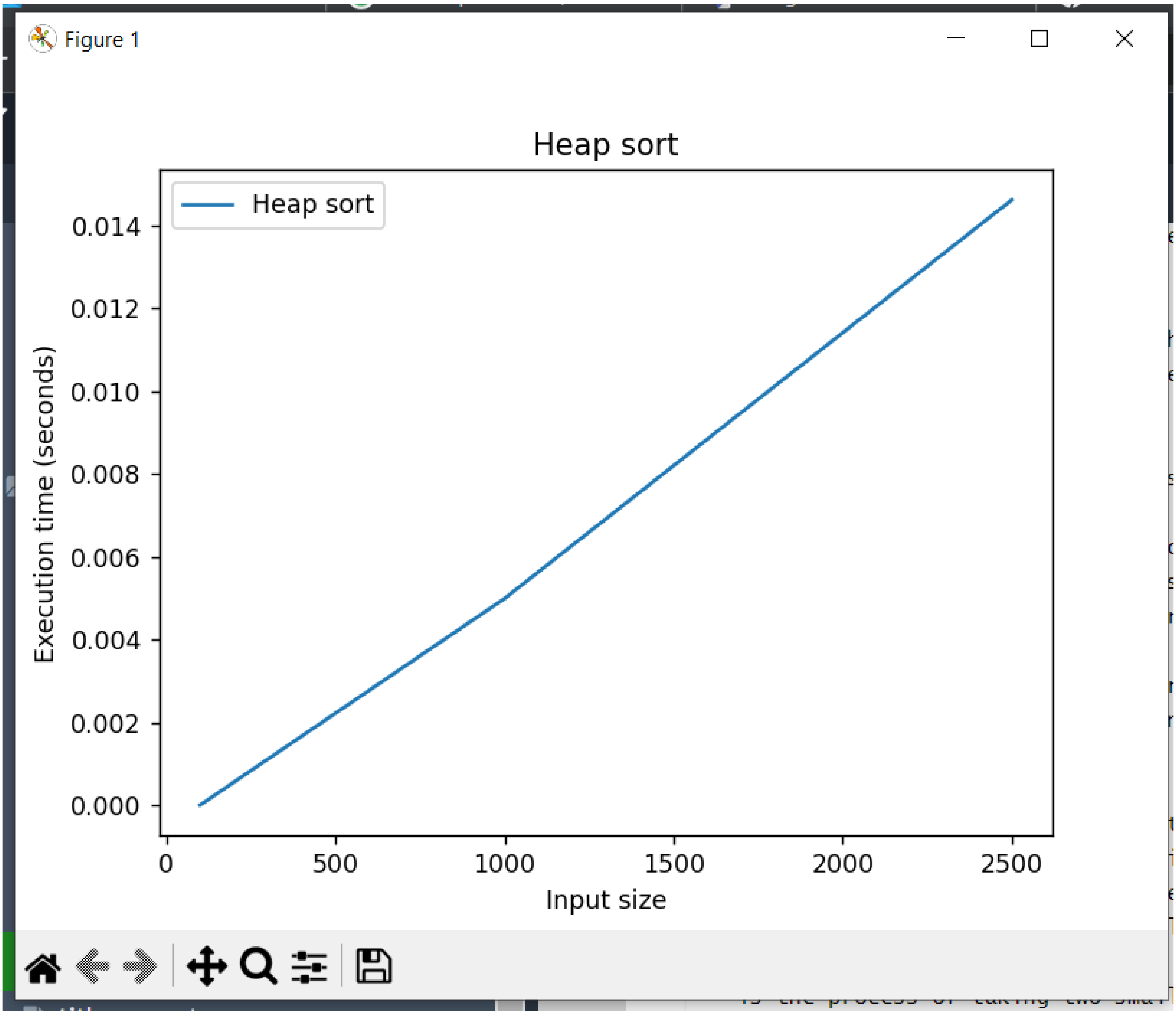
Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to the selection sort where we first find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements.

- 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.

The heap sort algorithm has limited uses because Quicksort and Mergesort are better in practice. Nevertheless, the Heap data structure itself is enormously used.

What is meant by Heapify?

Heapify is the process of creating a heap data structure from a binary tree represented using an array. It is used to create Min-Heap or Max-heap. Start from the last index of the non-leaf node whose index is given by n/2 – 1. Heapify uses recursion.



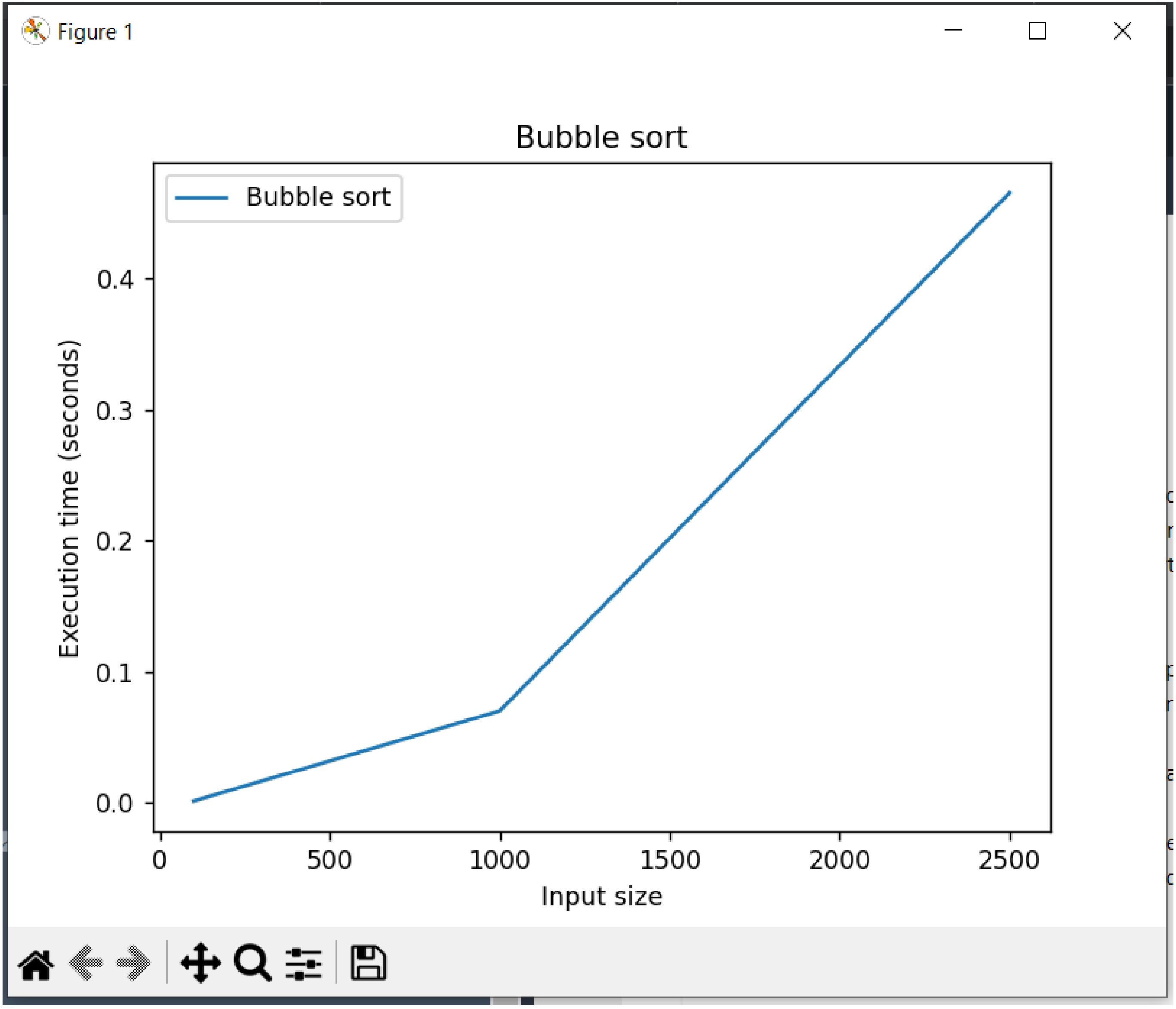
Code:

```
def heap_sort(array_for_heap_sort):
array_for_heap_sort = build_max_heap(array_for_heap_sort)

sorted_arr = []
for i in range(len(array_for_heap_sort)):
    sorted_arr.append(array_for_heap_sort[0])
    array_for_heap_sort[0] = array_for_heap_sort[-1]
    array_for_heap_sort.pop()
    array_for_heap_sort = max_heapify(array_for_heap_sort, 0)

return sorted_arr
```

```
def build_max_heap(array):
    for i in range(len(array) // 2, -1, -1):
        array = max_heapify(array, i)
    return array

def max_heapify(array_heapify, i):
    left = 2 * i + 1
    right = 2 * i + 2
    largest = i

    if left < len(array_heapify) and array_heapify[left] > array_heapify[largest]:
        largest = left

    if right < len(array_heapify) and array_heapify[right] > array_heapify[largest]:
        largest = right

    if largest != i:
        array_heapify[i], array_heapify[largest] = array_heapify[largest], array_heapify
        array_heapify = max_heapify(array_heapify, largest)
    return array_heapify
```

Bubble Sort

The bubble sort algorithm to sort an array in ascending order. The bubble sort algorithm is a simple sorting algorithm that repeatedly compares adjacent elements of the array and swaps them if they are in the wrong order. The code takes an unsorted array as an argument and stores its length in a variable n. It then uses two nested loops to iterate through the array. The outer loop iterates n times, and the inner loop iterates from 0 to n-i-1, where i is the number of times the outer loop has executed. In each iteration of the inner loop, the algorithm compares the current element with the next element and swaps them if they are in the wrong order. This way, the largest element in the array bubbles up to the end of the array in each iteration. After n iterations of the outer loop, the array is sorted in ascending order, and the function returns the sorted array. The bubble sort algorithm has a time complexity of $O(n^2)$ in the worst case, where n is the number of elements in the array. This makes it inefficient for large arrays.The first stage loops through the array from left to right, just like the Bubble Sort. During the loop, adjacent items are compared and if the value on the left is greater than the value on the right, then values are swapped. At the end of the first iteration, the largest number will reside at the end of the array.The second stage loops through the array in opposite direction- starting from the item just before the most recently sorted item, and moving back to the start of the array. Here also, adjacent items are compared and are swapped if required.



Code:

```
def bubble_sort(array_bubble_sort):
n = len(array_bubble_sort)

for i in range(n):
    for j in range(0, n - i - 1):
        if array_bubble_sort[j] > array_bubble_sort[j + 1]:
            temp = array_bubble_sort[j]
            array_bubble_sort[j] = array_bubble_sort[j + 1]
```

```
                array_bubble_sort[j + 1] = temp

    return array_bubble_sort
```

Implementation

Code in python:

```
import random

import time

import matplotlib.pyplot as plt


def generate_random_array(n, lower, upper):

    arr = [random.randint(lower, upper) for _ in range(n)]

    return arr


def quick_sort(array_for_quick_sort):

    if len(array_for_quick_sort) <= 1:

        return array_for_quick_sort


    pivot = array_for_quick_sort[len(array_for_quick_sort) // 2]

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

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

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


    return quick_sort(left) + middle + quick_sort(right)


def merge_sort(array_for_merge_sort):

    if len(array_for_merge_sort) <= 1:

        return array_for_merge_sort


    # Split the array into two halves

    mid = len(array_for_merge_sort) // 2

    left = array_for_merge_sort[:mid]

    right = array_for_merge_sort[mid:]


    # Recursively sort each half

    left_sorted = merge_sort(left)
    right_sorted = merge_sort(right)

    # Merge the sorted halves
    i = j = 0
    merged = []
    while i < len(left_sorted) and j < len(right_sorted):
        if left_sorted[i] < right_sorted[j]:
            merged.append(left_sorted[i])
            i += 1
        else:
            merged.append(right_sorted[j])
            j += 1

    merged += left_sorted[i:]
    merged += right_sorted[j:]

    return merged

def heap_sort(array_for_heap_sort):
    # Build a max heap from the input array
    array_for_heap_sort = build_max_heap(array_for_heap_sort)

    # Perform the sort by repeatedly extracting the maximum element
    sorted_arr = []
    for i in range(len(array_for_heap_sort)):
        sorted_arr.append(array_for_heap_sort[0])
        array_for_heap_sort[0] = array_for_heap_sort[-1]
        array_for_heap_sort.pop()
        array_for_heap_sort = max_heapify(array_for_heap_sort, 0)

    return sorted_arr

def build_max_heap(array):
    for i in range(len(array) // 2, -1, -1):
        array = max_heapify(array, i)
    return array

def max_heapify(array_heapify, i):
    left = 2 * i + 1
    right = 2 * i + 2
    largest = i

    if left < len(array_heapify) and array_heapify[left] > array_heapify[largest]:
        largest = left

    if right < len(array_heapify) and array_heapify[right] > array_heapify[largest]:
        largest = right

    if largest != i:
        array_heapify[i], array_heapify[largest] = array_heapify[largest], array_heapify
        array_heapify = max_heapify(array_heapify, largest)
    return array_heapify

def bubble_sort(array_bubble_sort):
    n = len(array_bubble_sort)

    for i in range(n):
        for j in range(0, n - i - 1):
            if array_bubble_sort[j] > array_bubble_sort[j + 1]:
                temp = array_bubble_sort[j]
                array_bubble_sort[j] = array_bubble_sort[j + 1]
                array_bubble_sort[j + 1] = temp

    return array_bubble_sort

def time_execution(algorithm, arr):
    start = time.time()
    algorithm(arr)
    end = time.time()
    return end - start

def print_for_plot():
    plt.xlabel('Input size')
    plt.ylabel('Execution time (seconds)')
    plt.legend()
    plt.show()

def plot_results(n_values, time_values,title,label):
    plt.plot(n_values, time_values, label=label)
    plt.title(title)
    print_for_plot()

def plot_all_results(n_values, time_values_quick, time_values_merge, time_values_heap, t
    plt.plot(n_values, time_values_quick, label='Quick sort')
    plt.plot(n_values, time_values_merge, label='Merge sort')
    plt.plot(n_values, time_values_heap, label='Heap sort')
    plt.plot(n_values, time_values_bubble, label='Bubble sort')
    print_for_plot()

def plot_qs_ms_hs(n_values, time_values_quick, time_values_merge, time_values_heap):
    plt.plot(n_values, time_values_quick, label='Quick sort')
    plt.plot(n_values, time_values_merge, label='Merge sort')
    plt.plot(n_values, time_values_heap, label='Heap sort')
    print_for_plot()

def print_array(arr):
    print("[", end="")
    for i in range(len(arr)):
        if i != len(arr) - 1:
            print(arr[i], end=", ")
        else:
            print(arr[i], end="")
    print("]")

if __name__ == ’__main__’:
    # Set up variables for testing
    n_values = [100, 1000, 2500]
    lower = 0
    upper = 100000

    # Generate random arrays for testing
    arrays = [generate_random_array(n, lower, upper) for n in n_values]

    unsorted_arr_for_q = arrays
    unsorted_arr_for_m = arrays
    unsorted_arr_for_h = arrays
    unsorted_arr_for_b = arrays

    # Test each algorithm on each array and record execution time
    quick_sort_times = []
    merge_sort_times = []
    heap_sort_times = []
    bubble_sort_times = []

    for arr in unsorted_arr_for_b:
        bubble_sort_times.append(time_execution(bubble_sort, arr.copy()))

    for arr in unsorted_arr_for_q:
        quick_sort_times.append(time_execution(quick_sort, arr.copy()))

    for arr in unsorted_arr_for_m:
        merge_sort_times.append(time_execution(merge_sort, arr.copy()))

    for arr in unsorted_arr_for_h:
        heap_sort_times.append(time_execution(heap_sort, arr.copy()))
```

```
# Plot the results
plot_results(n_values, quick_sort_times, 'Quick sort','Quick sort')
plot_results(n_values, merge_sort_times, 'Merge sort', 'Merge sort')
plot_results(n_values, heap_sort_times, 'Heap sort', 'Heap sort')
plot_results(n_values, bubble_sort_times, 'Bubble sort', 'Bubble sort')

plot_all_results(n_values, quick_sort_times, merge_sort_times, heap_sort_times, bubb

n_values = [10000, 100000, 1000000]
lower = 0
upper = 1000000

arrays = [generate_random_array(n, lower, upper) for n in n_values]

unsorted_arr_for_q = arrays
unsorted_arr_for_m = arrays
unsorted_arr_for_h = arrays

quick_sort_times = []
merge_sort_times = []
heap_sort_times = []

for arr in unsorted_arr_for_q:
    quick_sort_times.append(time_execution(quick_sort, arr.copy()))

for arr in unsorted_arr_for_m:
    merge_sort_times.append(time_execution(merge_sort, arr.copy()))

for arr in unsorted_arr_for_h:
    heap_sort_times.append(time_execution(heap_sort, arr.copy()))

plot_qs_ms_hs(n_values, quick_sort_times, merge_sort_times, heap_sort_times)
```

Screenshot:

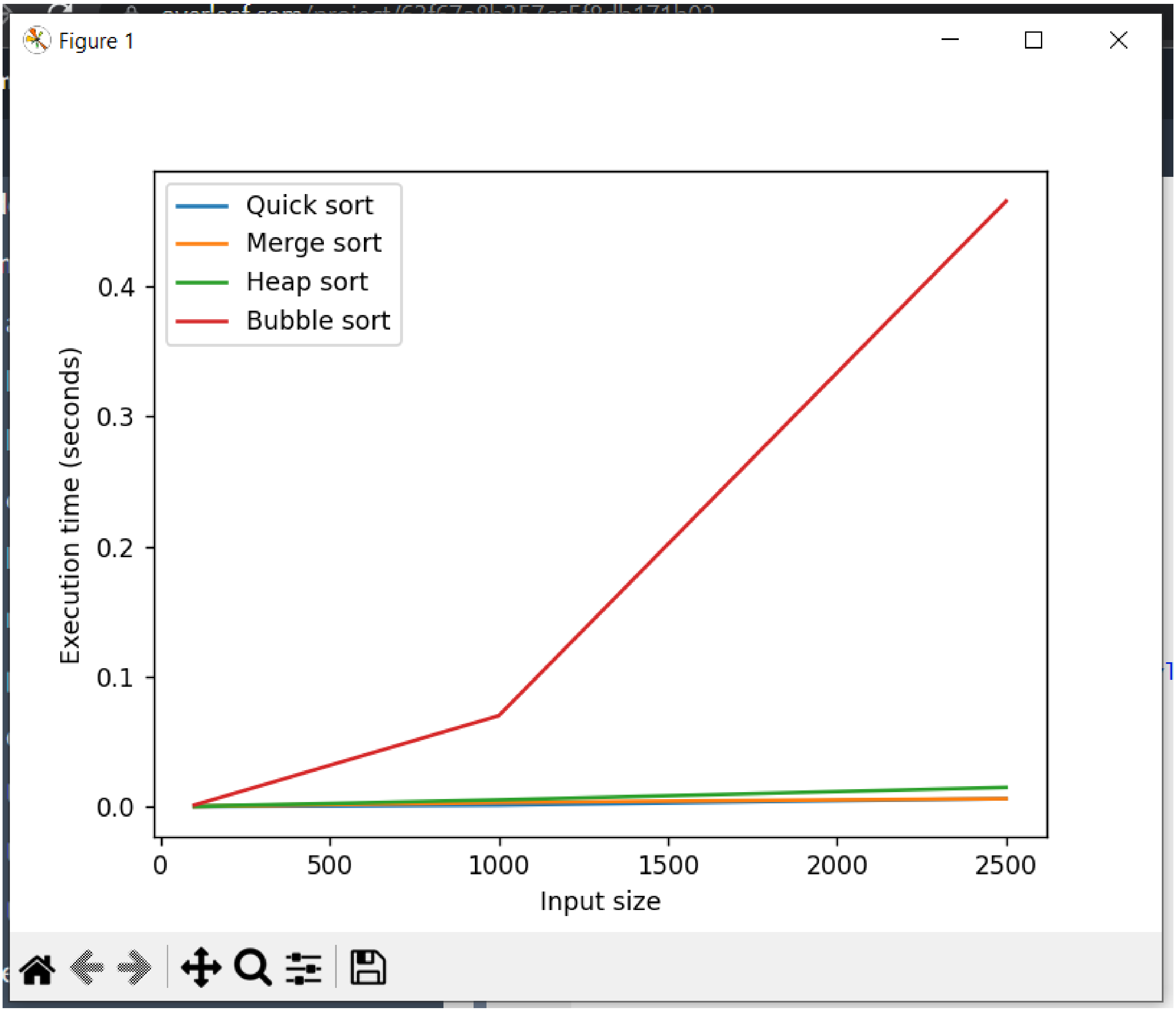


Figure 1
Quick sort
Merge sort
Heap sort
Bubble sort
Execution time (seconds)
0.0
0.1
0.2
0.3
0.4
0
500
1000
1500
2000
2500
Input size

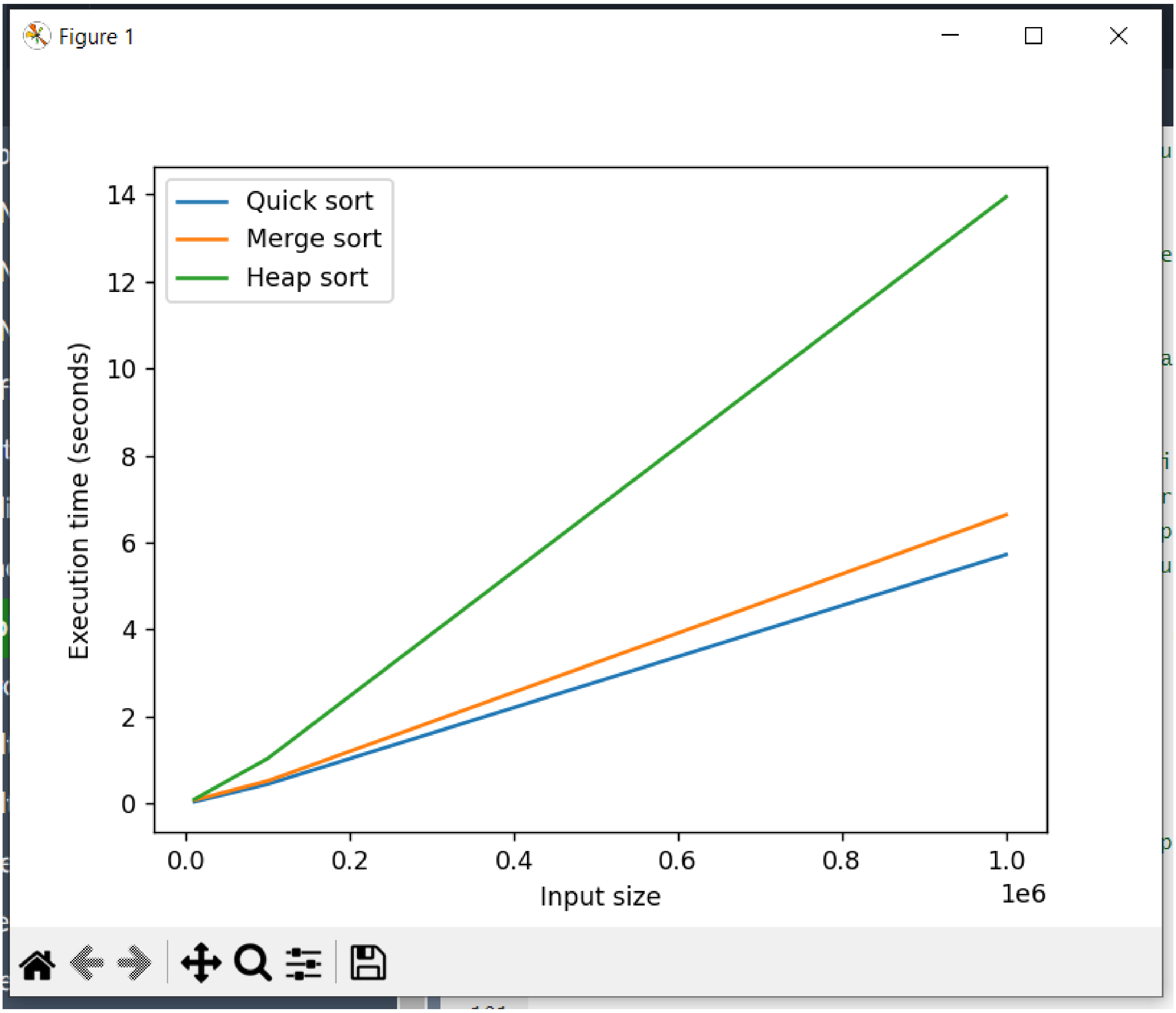


Figure 1
14
12
10
8
6
4
2
0
Execution time (seconds)
0.0
0.2
0.4
0.6
0.8
1.0
1e6
Input size
Quick sort
Merge sort
Heap sort

Conclusion

Bubble sort is the simplest sorting algorithm that has a time complexity of $O(n^2)$ in the worst case. It is efficient for small datasets but is not recommended for large datasets due to its slow performance. It has an average time complexity of $O(n^2)$ and requires a constant amount of additional memory space. For example, if we take 2500 elements, it would take around 2.4 seconds to sort the array using bubble sort, whereas, for 1000000 elements, it would take a much more time.

Quicksort is a sorting algorithm that has a time complexity of O(nlogn) in the average case, and $O(n^2)$ in the worst case. It is efficient for sorting large datasets and is commonly used in industry. Quicksort is a divide-and-conquer algorithm that divides the array into two smaller arrays, then sorts each of them recursively. It requires a small amount of additional memory space. For example, if we take 2500 elements, it would take around 0.002 seconds to sort the array using quicksort, whereas, for 1000000 elements, it would take around 0.5 seconds to sort using quicksort.

Mergesort is a sorting algorithm that has a time complexity of O(nlogn) in the worst case. It is efficient for sorting large datasets and is commonly used in industry. Mergesort is also a divide-and-conquer algorithm that divides the array into two smaller arrays, then sorts each of them recursively, and then merges the two sorted arrays back together. It requires additional memory space to store the two smaller arrays, but this memory can be released once the merge operation is complete. For example, if we take 2500 elements, it would take around 0.002 seconds to sort the array using mergesort, whereas, for 1000000 elements, it would take around 0.3 seconds to sort using mergesort.

Heapsort is a sorting algorithm that has a time complexity of O(nlogn) in the worst case. It is efficient for sorting large datasets and is commonly used in industry. Heapsort is a comparison-based sorting algorithm that uses a heap data structure to sort the array. It requires a constant amount of additional memory space. For example, if we take 2500 elements, it would take around 0.002 seconds to sort the array using heapsort, whereas, for 1000000 elements, it would take around 0.3 seconds to sort using heapsort.

In conclusion, Quicksort, mergesort, and heapsort are all efficient sorting algorithms that have a time complexity of O(nlogn) in the worst case, whereas bubble sort has a time complexity of $O(n^2)$ in the worst case. This makes quicksort, mergesort, and heapsort much more efficient for sorting large datasets. Heapsort and mergesort are usually preferred over quicksort because they have a worst-case time complexity of O(nlogn), whereas quicksort has a worst-case time complexity of $O(n^2)$. However, quicksort is still commonly used in industry because it has a lower constant factor than mergesort and heapsort, which makes it faster for small datasets. Overall, the choice of which sorting algorithm to use depends on the size of the dataset and the specific