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 Nistor Stefan

Verified:

asist.univ. Fiștic Cristofor

Chișinău, 2023

Content

Introduction

Sorting algorithms are a fundamental concept in computer science and are essential for organizing and processing data efficiently. In computer science, sorting refers to the process of rearranging a collection of items in a specific order. Sorting algorithms are used to sort data structures like arrays, lists, and trees, and the efficiency of the sorting algorithm is determined by the number of comparisons and swaps it takes to sort the data.

There are numerous sorting algorithms, each with its own strengths and weaknesses. Some of the most popular sorting algorithms include bubble sort, selection sort, insertion sort, merge sort, quicksort, and heapsort. Each algorithm has its own approach to sorting data, with different time and space complexity.

Understanding sorting algorithms is crucial for computer scientists, as efficient sorting can improve the performance of various applications such as databases, search engines, and operating systems. Moreover, a deeper understanding of sorting algorithms can help you become a better programmer and problem-solver, regardless of the field you're in. Sorting algorithms can be categorized as either internal or external, depending on how the data is stored during the sorting process. Internal sorting algorithms sort 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 main memory and must be sorted on disk or other external storage media.

Efficient sorting algorithms are vital for handling large datasets, where the time and space complexity of the algorithm can make a significant impact on the performance. There are various ways to measure the efficiency of sorting algorithms, including the number of comparisons and swaps, the time taken to sort the data, and the amount of memory used.

Sorting algorithms also have numerous real-world applications, from sorting names in a phone book to sorting large datasets in scientific research. They are used in various industries, including finance, healthcare, and engineering. Additionally, sorting algorithms are often used in combination with other algorithms to optimize complex operations, such as data compression and searching.

Overall, sorting algorithms play a crucial role in computer science and data processing. Understanding how they work and their different approaches can help you choose the most efficient algorithm for a given task, optimize the performance of your code, and improve your 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 partition(array, low, high):

# Choose the rightmost element as pivot
pivot = array[high]

# Pointer for greater element
i = low - 1

# Traverse through all elements
# compare each element with pivot
for j in range(low, high):
        if array[j] <= pivot:
            # If element smaller than pivot is found
            # swap it with the greater element pointed by i
            i = i + 1

            # Swapping element at i with element at j
            (array[i], array[j]) = (array[j], array[i])

    # Swap the pivot element with
    # e greater element specified by i
    (array[i + 1], array[high]) = (array[high], array[i + 1])

    # Return the position from where partition is done
    return i + 1

# Function to perform quicksort


def quick_sort(array, low, high):
    if low < high:

        # Find pivot element such that
        # element smaller than pivot are on the left
        # element greater than pivot are on the right
        pi = partition(array, low, high)

        # Recursive call on the left of pivot
        quick_sort(array, low, pi - 1)

        # Recursive call on the right of pivot
        quick_sort(array, pi + 1, high)
```

```
array = [10, 7, 8, 9, 1, 5]
quick_sort(array, 0, len(array) - 1)

print(f'Sorted array: {array}')
```

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 mergeSort(arr):
if len(arr) > 1:

     # Finding the mid of the array
    mid = len(arr)//2

    # Dividing the array elements
    L = arr[:mid]
```

```
        # into 2 halves
        R = arr[mid:]

        # Sorting the first half
        mergeSort(L)

        # Sorting the second half
        mergeSort(R)

        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

# Code to print the list
```

```
def printList(arr):
    for i in range(len(arr)):
        print(arr[i], end=" ")
    print()



if __name__ == '__main__':
    arr = [12, 11, 13, 5, 6, 7]
    print("Given array is", end="\n")
    printList(arr)
    mergeSort(arr)
    print("Sorted array is: ", end="\n")
    printList(arr)
```

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 heapify(arr, N, i):
largest = i  # Initialize largest as root
l = 2 * i + 1     # left = 2*i + 1
```

```
    r = 2 * i + 2     # right = 2*i + 2

    # See if left child of root exists and is
    # greater than root
    if l < N and arr[largest] < arr[l]:
        largest = l

    # See if right child of root exists and is
    # greater than root
    if r < N and arr[largest] < arr[r]:
        largest = r

    # Change root, if needed
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]  # swap

        # Heapify the root.
        heapify(arr, N, largest)

# The main function to sort an array of given size


def heapSort(arr):
    N = len(arr)

    # Build a maxheap.
    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)
if __name__ == '__main__':
    arr = [12, 11, 13, 5, 6, 7]

    # Function call
    heapSort(arr)
    N = len(arr)

    print("Sorted array is")
    for i in range(N):
        print("%d" % arr[i], end=" ")
```

Bubble Sort

Bubble Sort is a simple sorting algorithm that works by repeatedly swapping adjacent elements if they are in the wrong order. It is called ”bubble sort” because the smaller elements ”bubble” to the top of the list in each iteration. Bubble Sort is easy to implement and understand, but it is relatively slow for large data sets. It has a worst-case time complexity of O(n2), where n is the number of elements in the array to be sorted.

Bubble Sort is most commonly used for educational purposes or in cases where the input data set is very small. It is rarely used in practice due to its slow performance compared to more efficient sorting algorithms, such as Quick Sort, Merge Sort, or Heap Sort.

Bubble Sort is stable, meaning that it preserves the relative order of equal elements in the input array. It is also an in-place sorting algorithm, meaning that it does not require any extra memory to store intermediate results. However, it has a relatively high number of comparisons and swaps, which makes it inefficient for large data sets. Algorithm:

1. 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.
2. 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(arr):
n = len(arr)
for i in range(n):
    for j in range(n - i - 1):
        if arr[j] > arr[j+1]:
            arr[j], arr[j+1] = arr[j+1], arr[j]
return arr
```

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[i
        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, tim
    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, bubble

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:

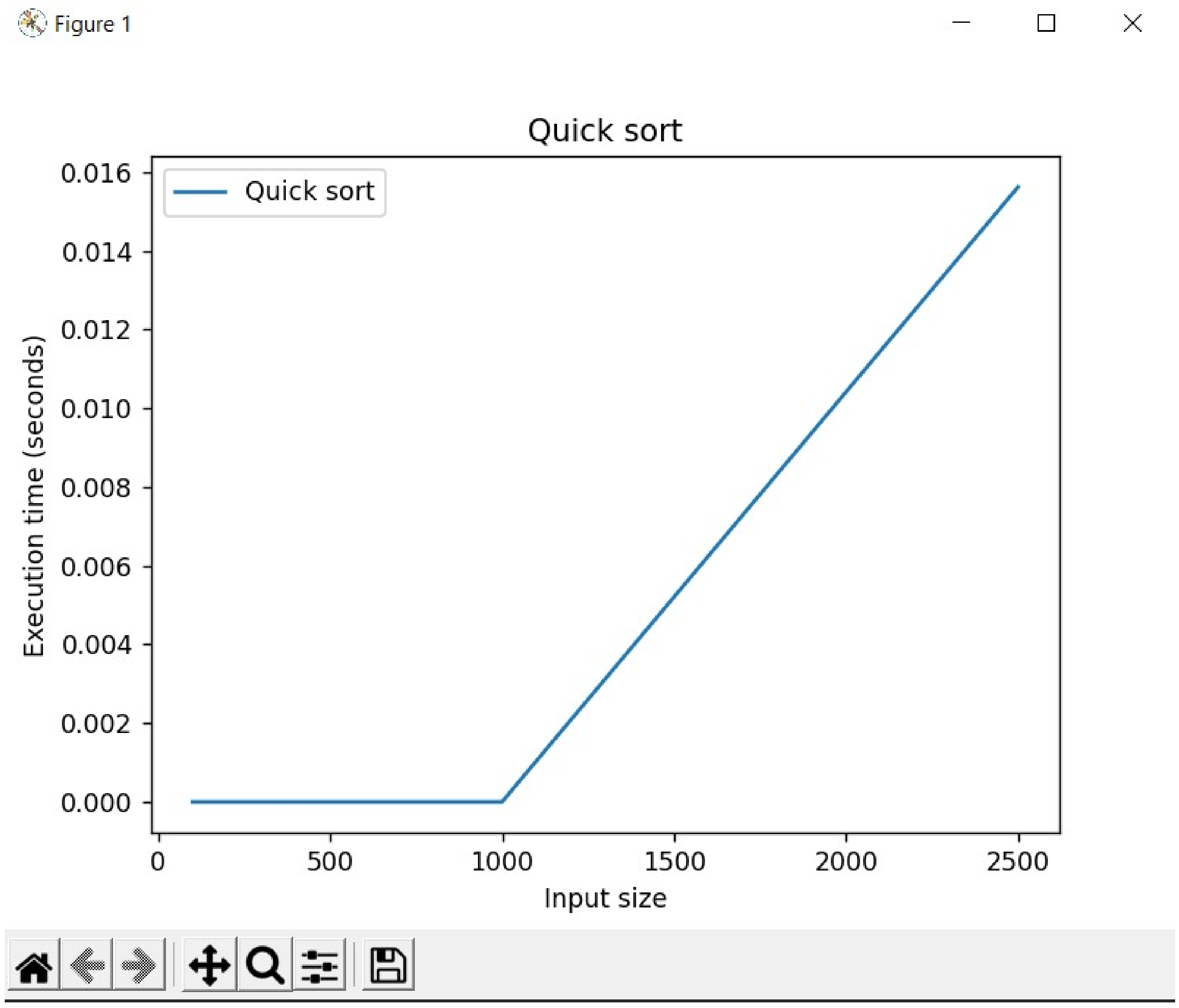
Figure 1
Quick sort
Quick sort
Execution time (seconds)
0.016
0.014
0.012
0.010
0.008
0.006
0.004
0.002
0.000
0
500
1000
1500
2000
2500
Input size

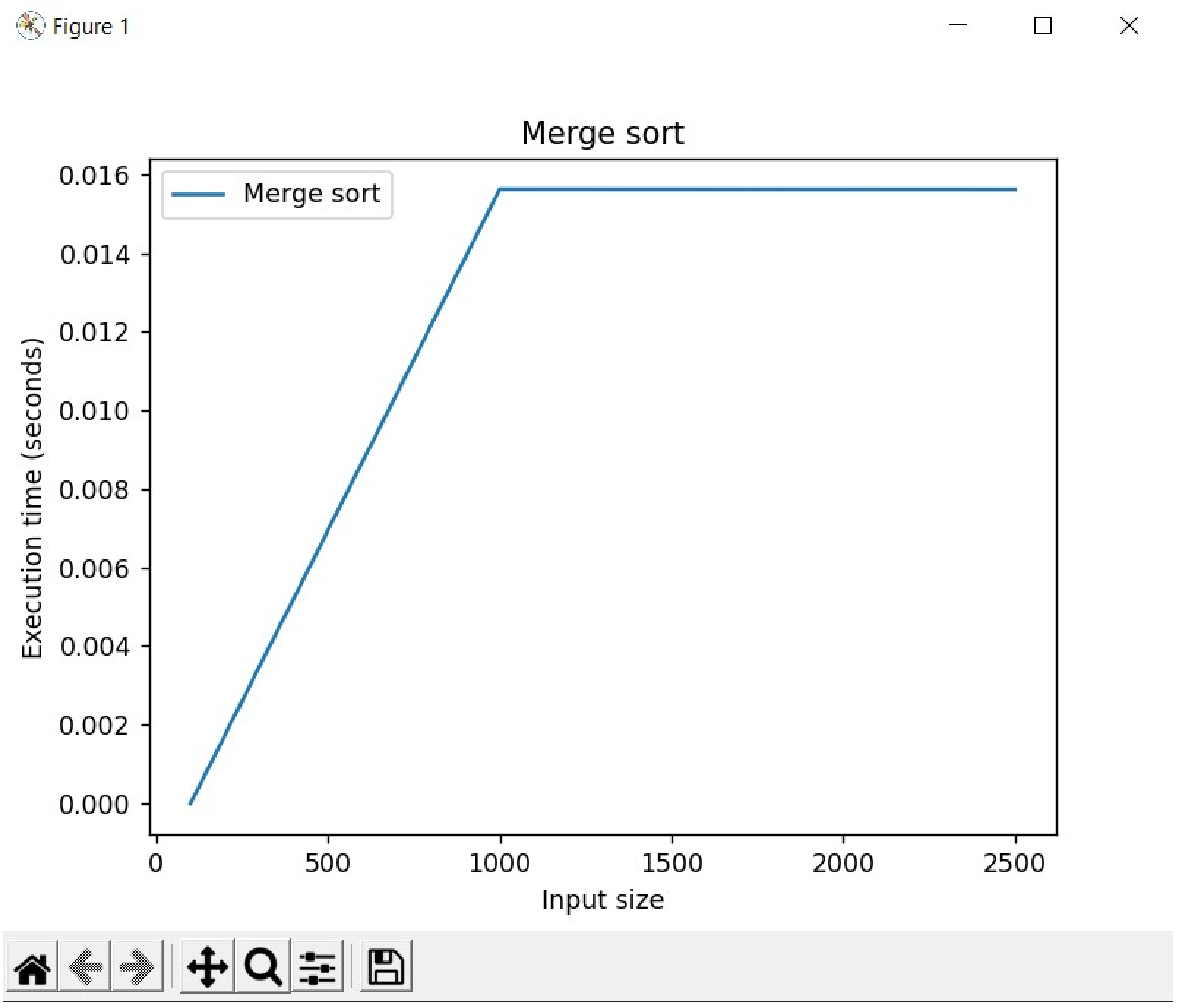
Figure 1
Merge sort
Merge sort
Execution time (seconds)
0.016
0.014
0.012
0.010
0.008
0.006
0.004
0.002
0.000
0
500
1000
1500
2000
2500
Input size

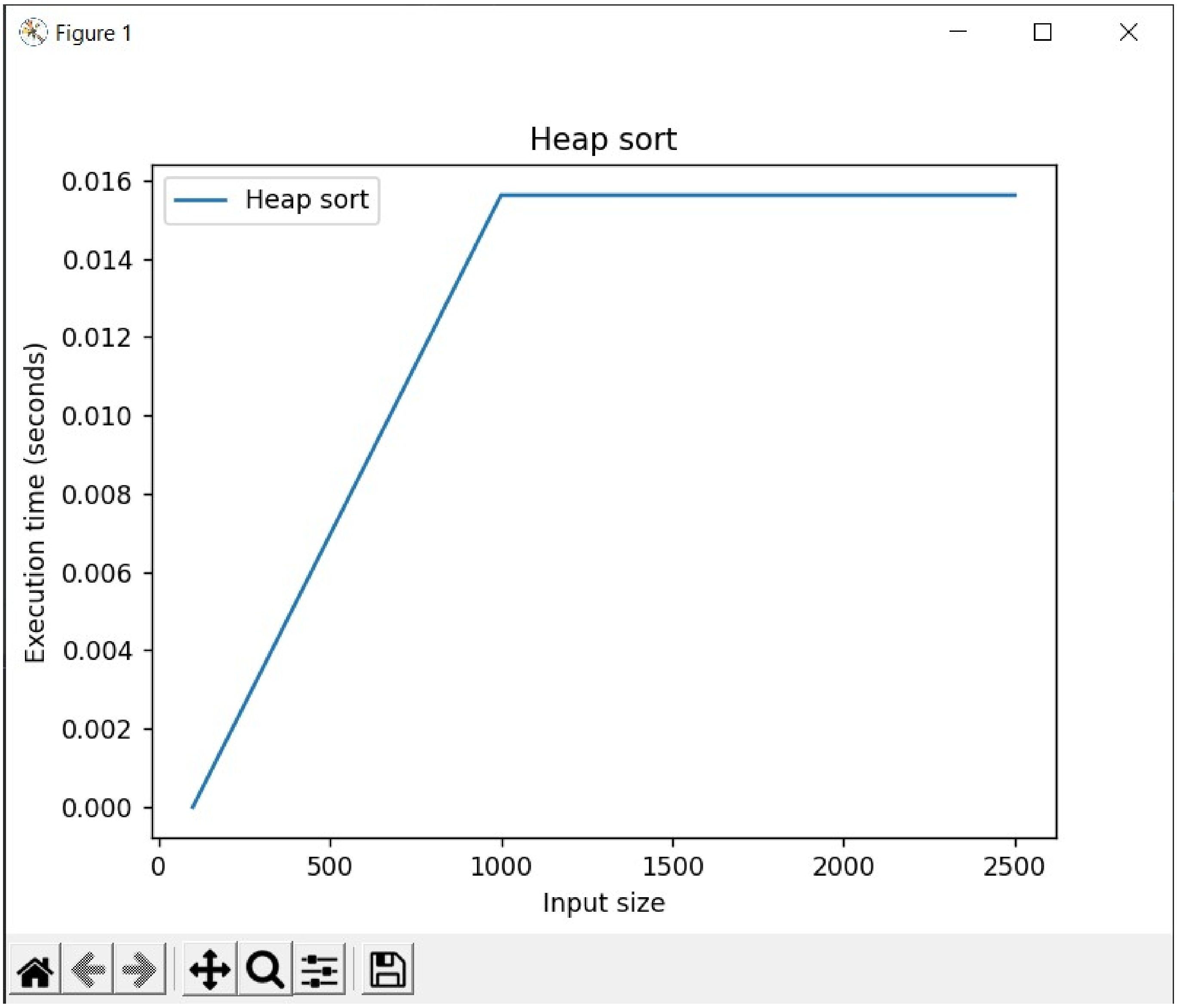
Figure 1
Heap sort
Heap sort
Execution time (seconds)
0.016
0.014
0.012
0.010
0.008
0.006
0.004
0.002
0.000
0
500
1000
1500
2000
2500
Input size

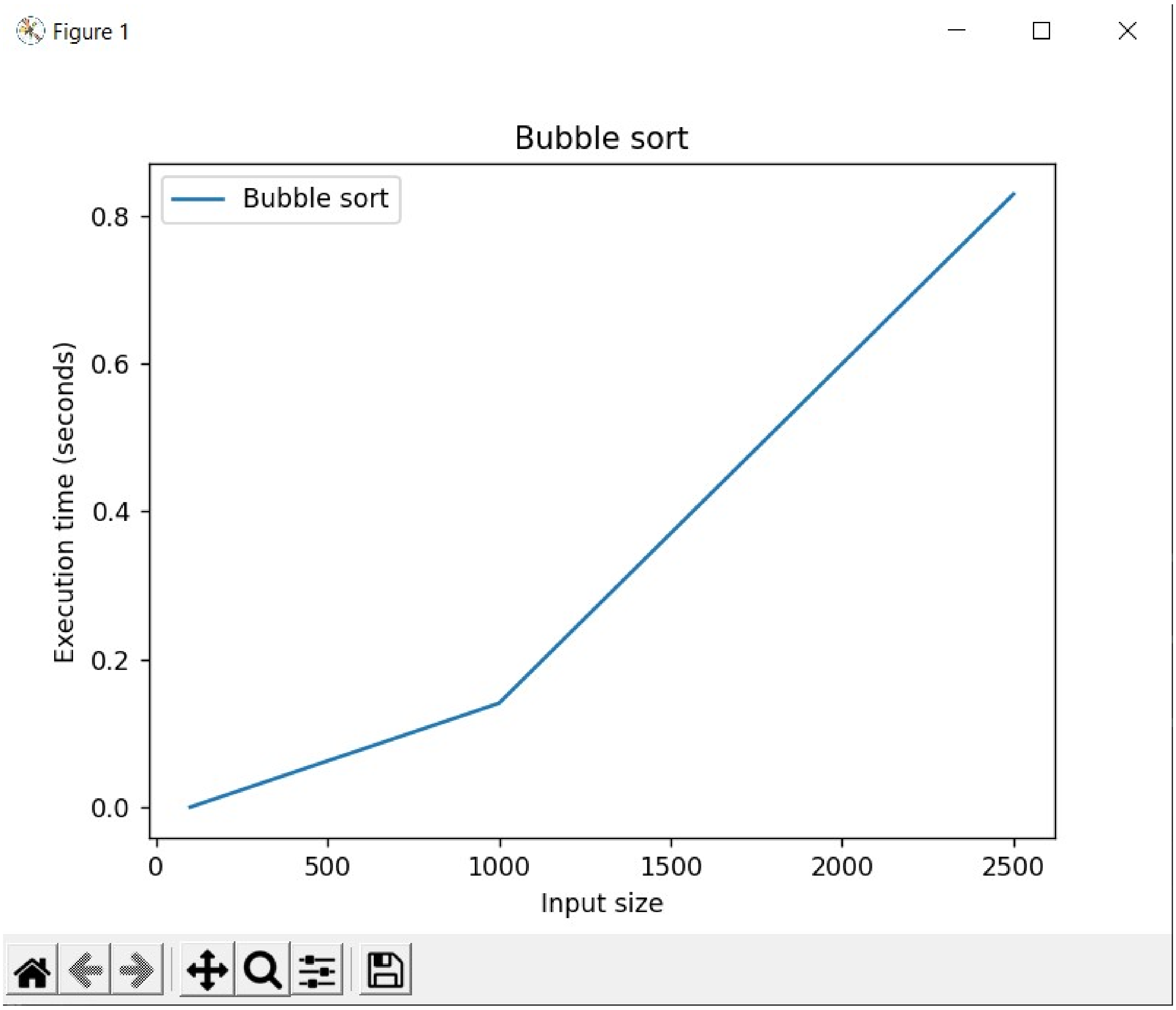
Figure 1
Bubble sort
Bubble sort
0.8
0.6
0.4
0.2
0.0
Execution time (seconds)
0
500
1000
1500
2000
2500
Input size

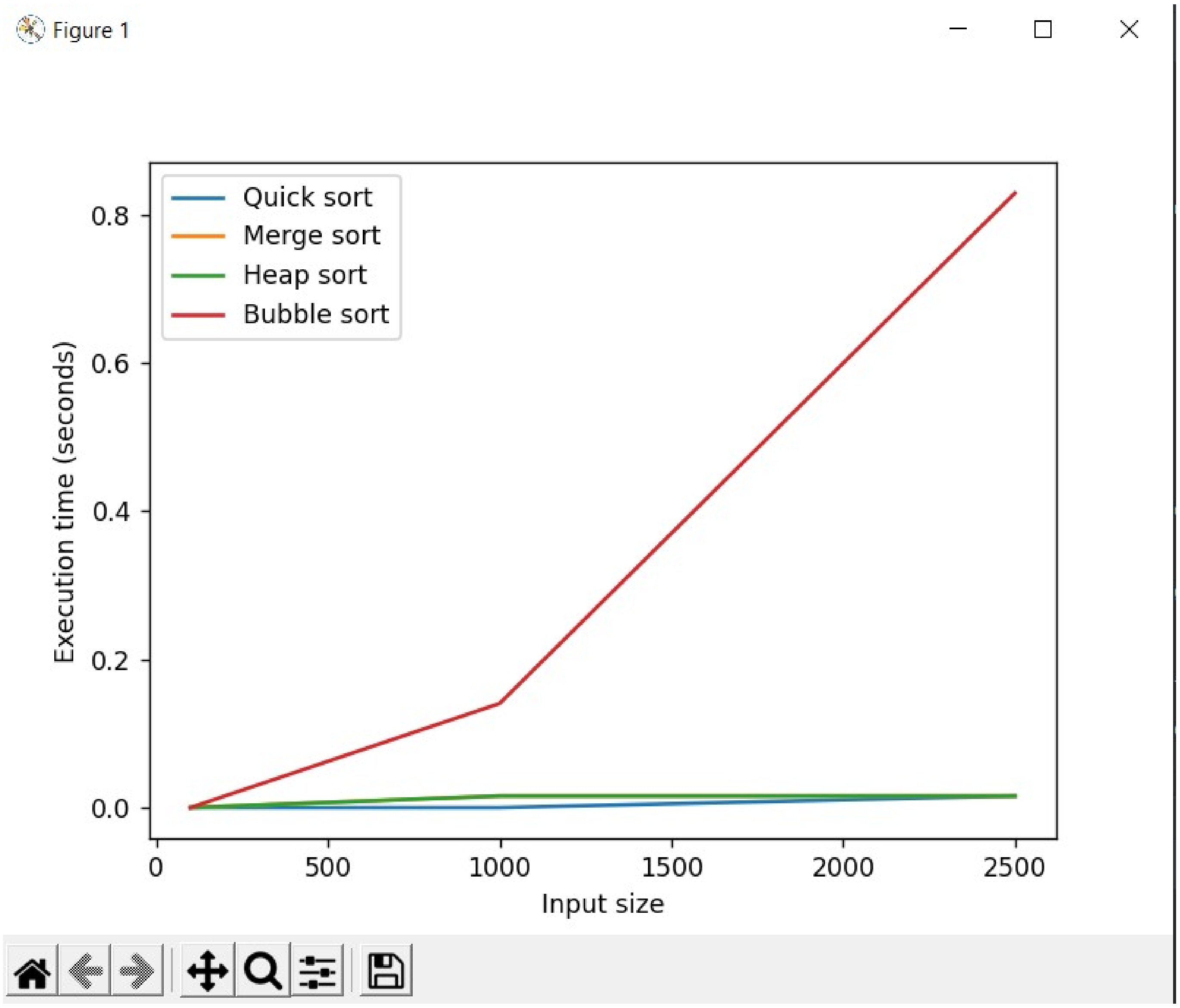
Figure 1
Quick sort
Merge sort
Heap sort
Bubble sort
Execution time (seconds)
0.0
0.2
0.4
0.6
0.8
0
500
1000
1500
2000
2500
Input size

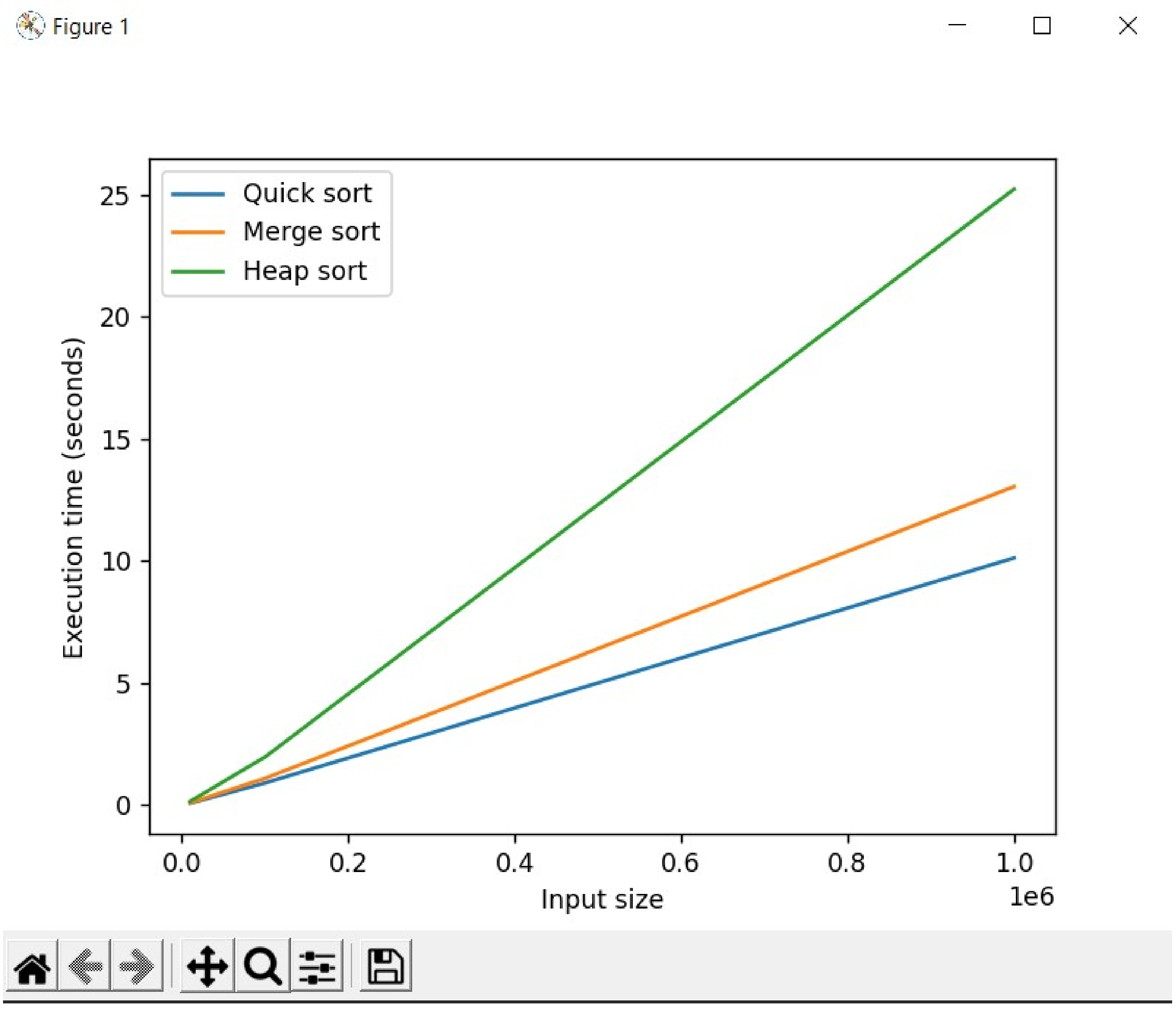
Figure 1
Quick sort
Merge sort
Heap sort
Execution time (seconds)
25
20
15
10
5
0
0.0
0.2
0.4
0.6
0.8
1.0
1e6
Input size

Conclusion

In conclusion, when sorting an array with 500,000 elements, the relative performance of quicksort, mergesort, heapsort, and cocktail sort can depend on various factors such as the input data, the hardware and software environment, and the implementation details.

Generally speaking, quicksort and mergesort have an average time complexity of O(n log n) and tend to perform well in practice, with quicksort being often preferred due to its lower constant factors and better cache locality. Heapsort, with a time complexity of O(n log n) in the worst case, is a good option when stable sorting is not required, and space complexity is a concern. Cocktail sort, while having a similar time complexity to bubblesort, can be more efficient due to its bidirectional scanning, but it is not widely used due to its complexity and relative slowness.

Therefore, if you are looking for an efficient and versatile sorting algorithm for large arrays, quicksort is a safe and popular choice, but you may also consider mergesort or heapsort depending on your specific requirements and constraints. It's always good to benchmark and profile different algorithms on your actual data and hardware setup to make an informed decision.

Overall, when choosing a sorting algorithm for a large array, you should consider factors such as the input data, the hardware and software environment, and the specific requirements and constraints of your application. You should also benchmark and profile different algorithms on your actual data and hardware setup to make an informed decision. In general, quicksort, mergesort, and heapsort are reliable and efficient sorting algorithms that can handle large datasets, while cocktail sort may be a good choice for smaller arrays or specialized use cases.