Ministerul Educaţiei și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică

Laboratory work 2:
Study and Empirical Analysis of Algorithms for Studying divide et impera

Elaborated:
st. gr. FAF-211 Echim Mihail

Verified:
asist. univ. Fiștic Cristofor

Chişinău - 2023

ALGORITHM ANALYSIS

Objective
Study and analyze different divide et impera sorting algorithms.

Tasks:

1. Implement 3 divide et impera sorting algorithms and 1 by choice;
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 the mathematical analysis of complexity is empirical analysis.

This may be useful for obtaining preliminary information on the complexity class of an algorithm; comparing the efficiency of two (or more) algorithms for solving the same problems; comparing the efficiency of several implementations of the same algorithm; obtaining information on the efficiency of implementing an algorithm on a particular computer.

In the empirical analysis of an algorithm, the following steps are usually followed:

1. The purpose of the analysis is established.
2. Choose the efficiency metric to be used (number of executions of an operation (s) or time execution of all or part of the algorithm.
3. The properties of the input data in relation to which the analysis is performed are established (data size or specific properties).
4. The algorithm is implemented in a programming language.
5. Generating multiple sets of input data.
6. Run the program for each input data set.
7. The obtained data are analyzed.

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 behavior 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:

The divide-and-conquer paradigm is often used to find an optimal solution of a problem. Its basic idea is to decompose a given problem into two or more similar, but simpler, subproblems, to solve them in turn, and to compose their solutions to solve the given problem. Problems of sufficient simplicity are solved directly. For example, to sort a given list of *n* natural numbers, split it into two lists of about *n*/2 numbers each, sort each of them in turn, and interleave both results appropriately to obtain the sorted version of the given list (see the picture). This approach is known as the merge sort algorithm.

The name "divide and conquer" is sometimes applied to algorithms that reduce each problem to only one sub-problem, such as the binary search algorithm for finding a record in a sorted list (or its analog in numerical computing, the bisection algorithm for root finding). These algorithms can be implemented more efficiently than general divide-and-conquer algorithms; in particular, if they use tail recursion, they can be converted into simple loops. Under this broad definition, however, every algorithm that uses recursion or loops could be regarded as a "divide-and-conquer algorithm". Therefore, some authors consider that the name "divide and conquer" should be used only when each problem may generate two or more subproblems. The name **decrease and conquer** has been proposed instead for the single-subproblem class.

An important application of divide and conquer is in optimization, where if the search space is reduced ("pruned") by a constant factor at each step, the overall algorithm has the same asymptotic complexity as the pruning step, with the constant depending on the pruning factor (by summing the geometric series); this is known as prune and search.

Comparison Metric:

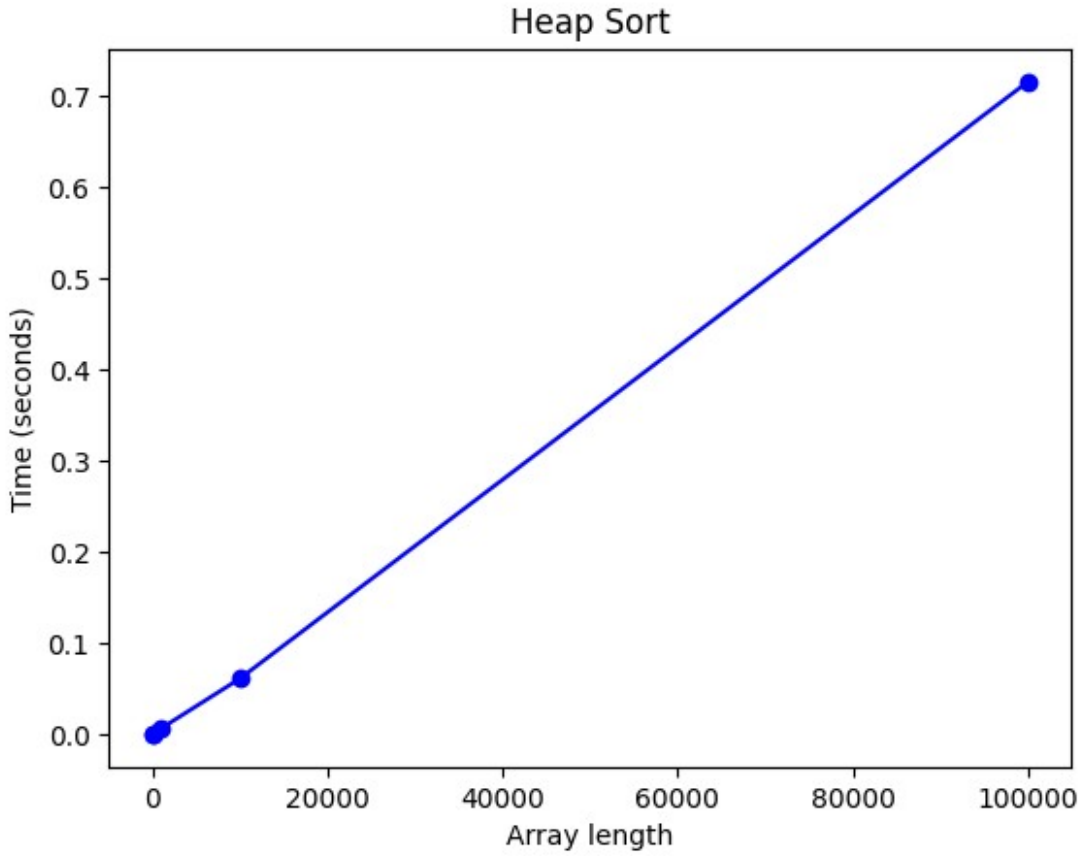
The comparison metric for this laboratory work will be considered the time of execution of each algorithm (T(n))

Input Format:

As input, each algorithm will receive arrays randomly filled with integers bigger than 0 and smaller than 101. The algorithms will be of sizes: 10, 100, 1000, 10000, and 100000. I tried using 10^6 as well, but most algorithms gave either maximum recursion error or memory error, so I decided to stop at 10^5.

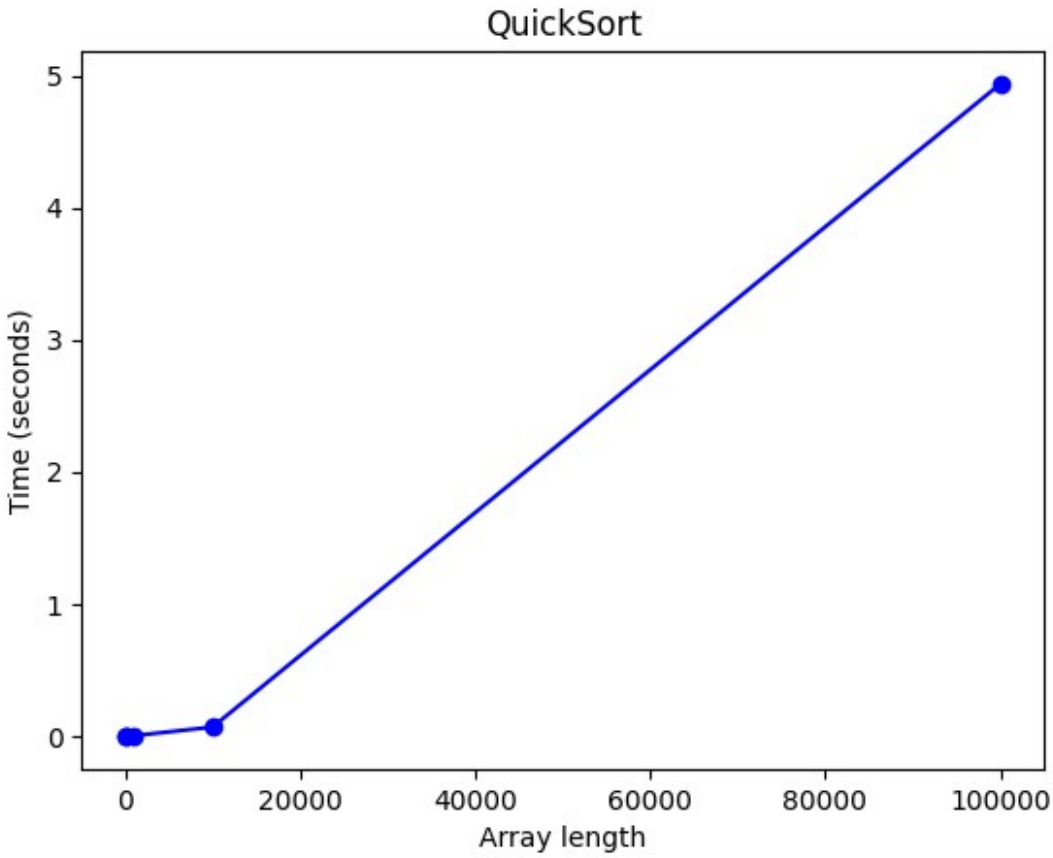
```python
from matplotlib import pyplot as plt
import random
import time
import sys

def heap_sort(arr):
    n = len(arr)

    # Build a max heap
    for i in range(n//2 - 1, -1, -1):
        heapify(arr, n, i)

    # Extract elements from the heap one by one
    for i in range(n-1, 0, -1):
        # Swap the root element (largest) with the last element
        arr[i], arr[0] = arr[0], arr[i]

        # Max heapify the reduced heap
        heapify(arr, i, 0)

def heapify(arr, n, i):
    # Initialize the largest as root
    largest = i
    left = 2*i + 1
    right = 2*i + 2

    # Check if left child of root exists and is greater than root
    if left < n and arr[left] > arr[largest]:
        largest = left

    # Check if right child of root exists and is greater than root
    if right < n and arr[right] > arr[largest]:
        largest = right

    # Change root, if needed
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        # Heapify the root
        heapify(arr, n, largest)


arr = []
times1 = []
cases = []
j = 1
for k in range(5):
    j *= 10
    for i in range(j):
        arr.append(random.randint(0, 100))
```

```
    start = time.time()
    heap_sort(arr)
    end = time.time()
    times1.append(end-start)
    cases.append(j)
# print("Sorted array is:", arr)

plt.plot(cases, times1, 'bo-')
plt.title('Heap Sort')
plt.xlabel('Array length')
plt.ylabel('Time (seconds)')
plt.show()
```




```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        less = [x for x in arr[1:] if x <= pivot]
        greater = [x for x in arr[1:] if x > pivot]
        return quicksort(less) + [pivot] + quicksort(greater)

arr = []
times2 = []
```

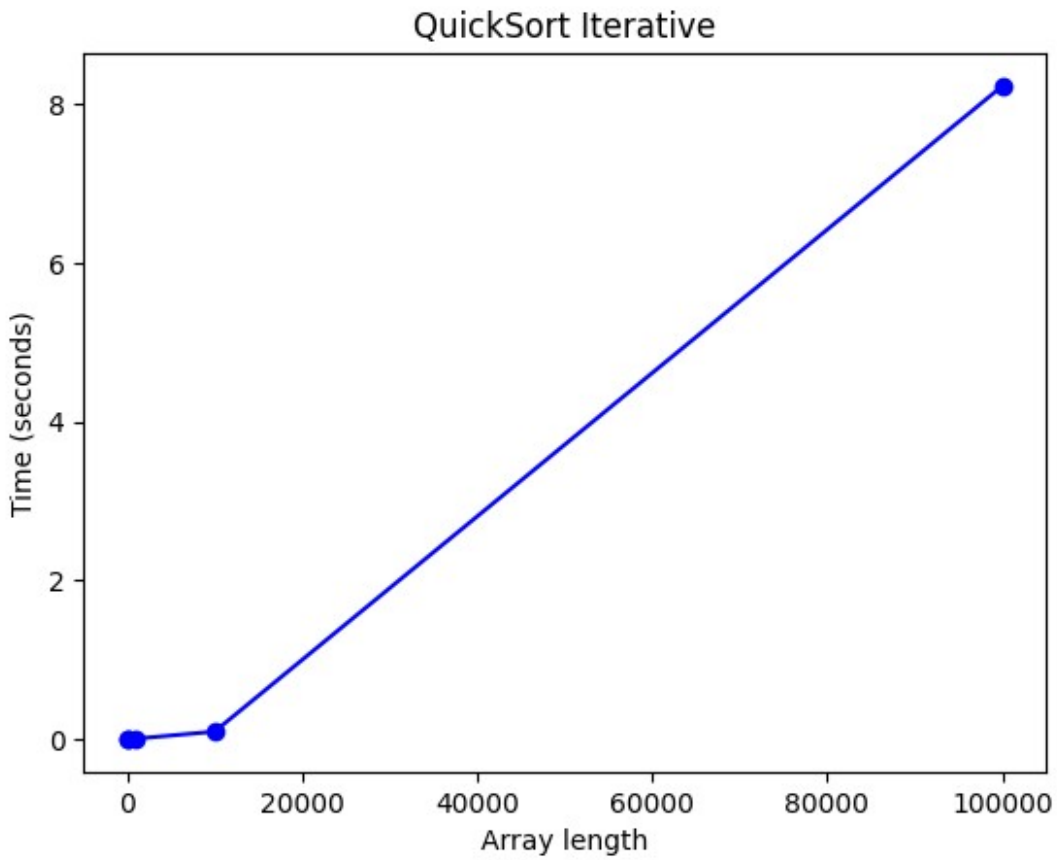
```
cases = []
j = 1
for k in range(5):
    j *= 10
    for i in range(j):
        arr.append(random.randint(0, 100))
    start = time.time()
    quicksort(arr)
    end = time.time()
    times2.append(end-start)
    cases.append(j)

plt.plot(cases, times2,'bo-')
plt.title('QuickSort')
plt.xlabel('Array length')
plt.ylabel('Time (seconds)')
plt.show()
```




```
# Python program for implementation of Quicksort

# This function is same in both iterative and recursive
def partition(arr,l,h):
    i = ( l - 1 )
    x = arr[h]
```

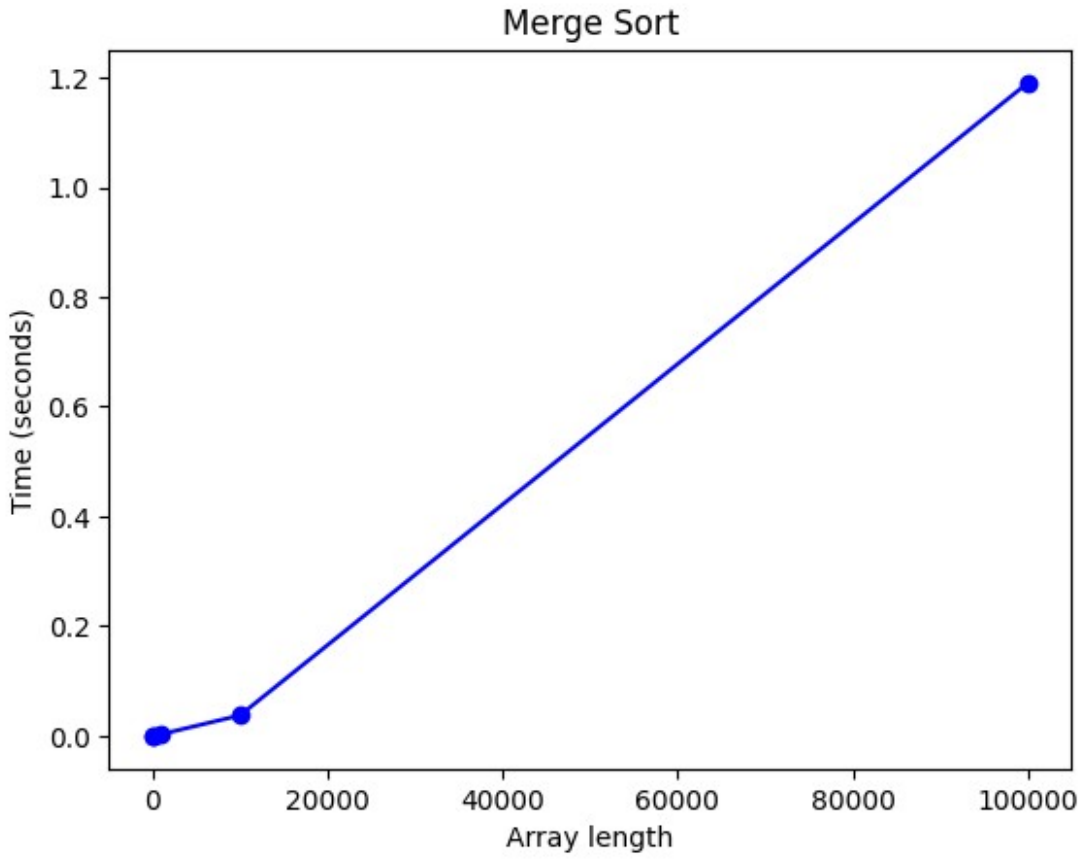
```
    for j in range(l , h):
        if   arr[j] <= x:

            # increment index of smaller element
            i = i+1
            arr[i],arr[j] = arr[j],arr[i]

    arr[i+1],arr[h] = arr[h],arr[i+1]
    return (i+1)

# Function to do Quick sort
# arr[] --> Array to be sorted,
# l  --> Starting index,
# h  --> Ending index
def quickSortIterative(arr,l,h):

    # Create an auxiliary stack
    size = h - l + 1
    stack = [0] * (size)

    # initialize top of stack
    top = -1

    # push initial values of l and h to stack
    top = top + 1
    stack[top] = l
    top = top + 1
    stack[top] = h

    # Keep popping from stack while is not empty
    while top >= 0:

        # Pop h and l
        h = stack[top]
        top = top - 1
        l = stack[top]
        top = top - 1

        # Set pivot element at its correct position in
        # sorted array
        p = partition( arr, l, h )

        # If there are elements on left side of pivot,
        # then push left side to stack
        if p-1 > l:
            top = top + 1
            stack[top] = l
            top = top + 1
            stack[top] = p - 1
```

```
        # If there are elements on right side of pivot,
        # then push right side to stack
        if p+1 < h:
            top = top + 1
            stack[top] = p + 1
            top = top + 1
            stack[top] = h


arr = []
times5 = []
cases = []
j = 1
for k in range(5):
    j *= 10
    for i in range(j):
        arr.append(random.randint(0, 100))
    start = time.time()
    quickSortIterative(arr, 0, j-1)
    end = time.time()
    times5.append(end-start)
    cases.append(j)

plt.plot(cases, times5,'bo-')
plt.title('QuickSort Iterative')
plt.xlabel('Array length')
plt.ylabel('Time (seconds)')
plt.show()
```

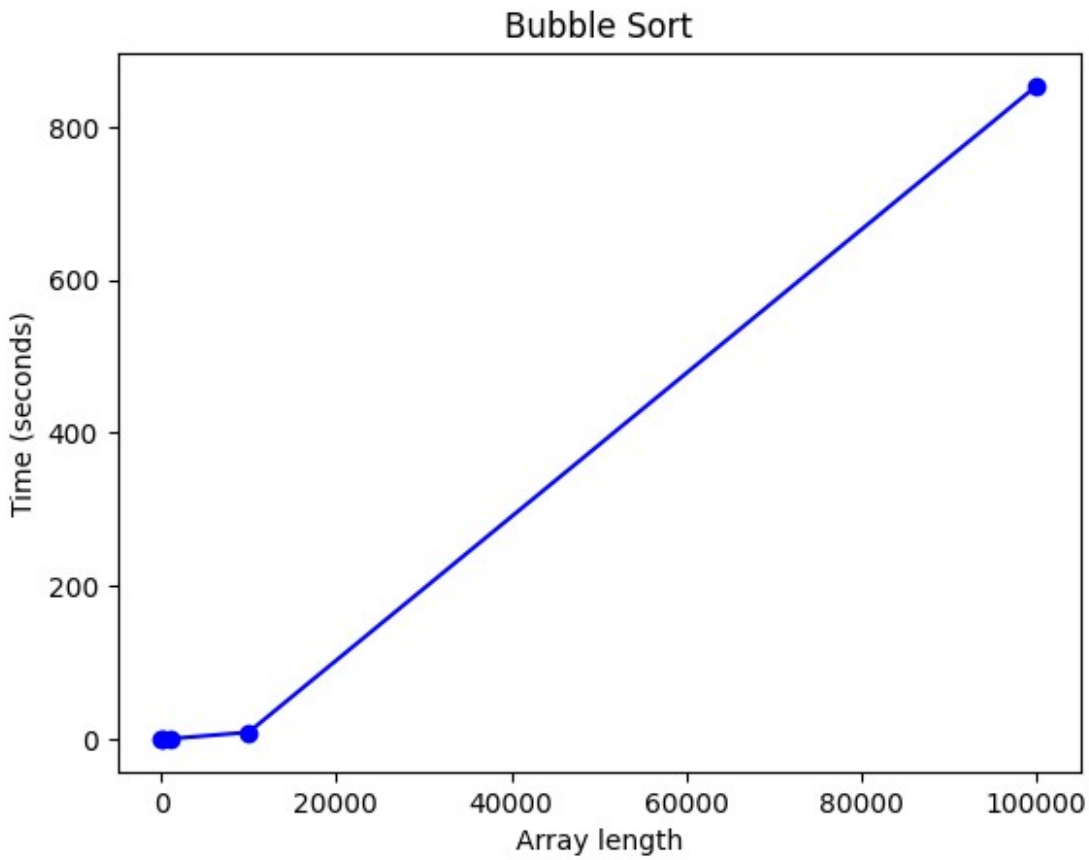



```python
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])

    return merge(left, right)

def merge(left, right):
    result = []

    while left and right:
        if left[0] <= right[0]:
            result.append(left.pop(0))
        else:
            result.append(right.pop(0))

    if left:
        result.extend(left)
    if right:
        result.extend(right)
```

```
    return result

arr = []
times3 = [] #change
cases = []
j = 1
for k in range(5):
    j *= 10
    for i in range(j):
        arr.append(random.randint(0, 100))
    start = time.time()
    merge_sort(arr)    #change
    end = time.time()
    times3.append(end-start) #change
    cases.append(j)

plt.plot(cases, times3,'bo-') #change
plt.title('Merge Sort') #change
plt.xlabel('Array length')
plt.ylabel('Time (seconds)')
plt.show()
```

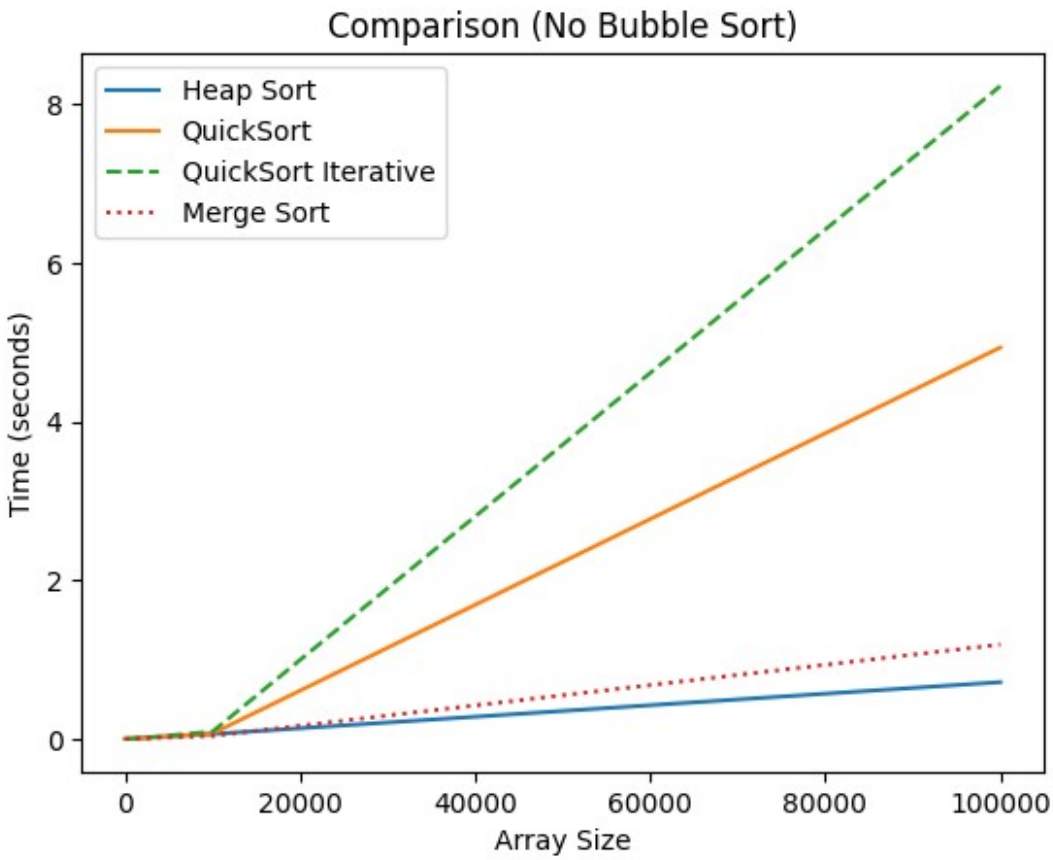


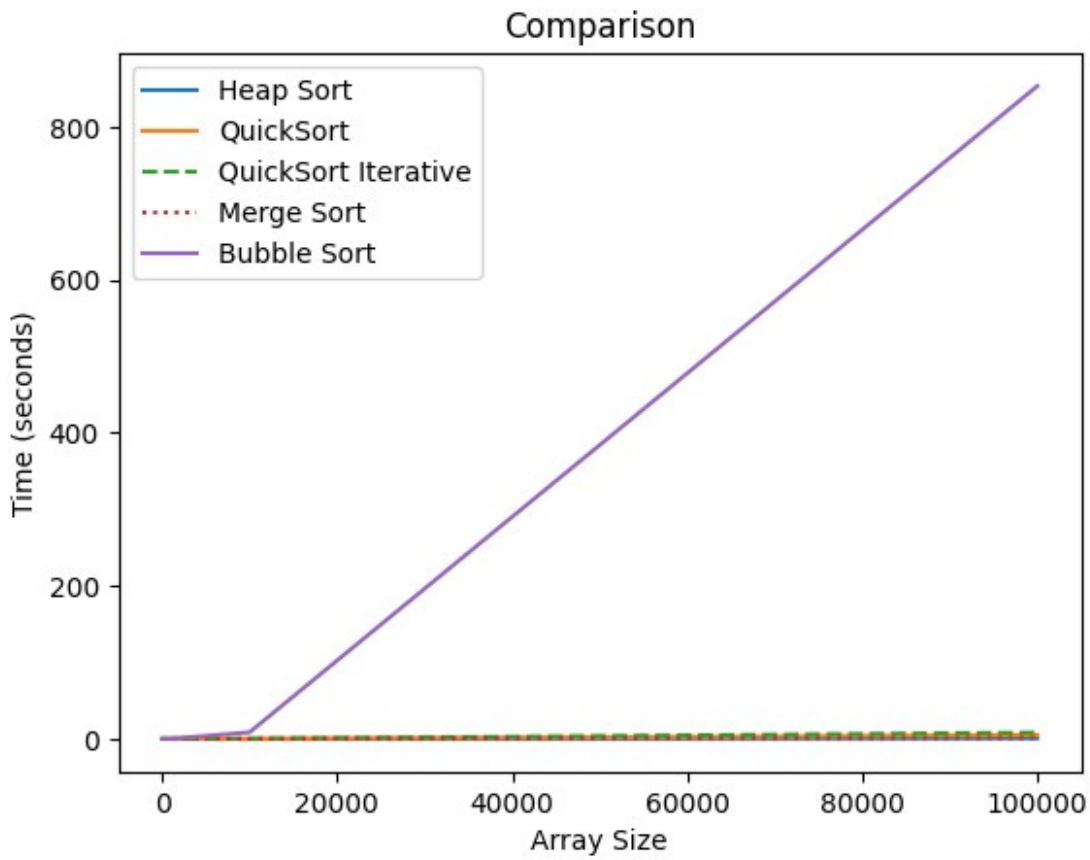
```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        # Flag to check if any swaps have been made
        swapped = False
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                # Swap the elements
                arr[j], arr[j+1] = arr[j+1], arr[j]
                # Set the flag to true
                swapped = True
        # If no swaps were made, the array is sorted and we can stop
iterating
        if not swapped:
            break
    return arr

arr = []
times4 = [] #change
cases = []
j = 1
for k in range(5):
    j *= 10
    for i in range(j):
        arr.append(random.randint(0, 100))
    start = time.time()
    bubble_sort(arr)    #change
    end = time.time()
    times4.append(end-start) #change
    cases.append(j)

plt.plot(cases, times4,'bo-') #change
plt.title('Bubble Sort') #change
plt.xlabel('Array length')
plt.ylabel('Time (seconds)')
plt.show()
```




```
plt.plot(cases,times1, label="Heap Sort")
plt.plot(cases,times2, label="QuickSort")
plt.plot(cases,times5 ,'--', label="QuickSort Iterative")
plt.plot(cases,times3, ':', label="Merge Sort")
plt.title('Comparison (No Bubble Sort)')
plt.xlabel('Array Size')
plt.ylabel('Time (seconds)')
plt.legend()
plt.show()
```

```
plt.plot(cases,times1, label="Heap Sort")
plt.plot(cases,times2, label="QuickSort")
plt.plot(cases,times5 ,'--', label="QuickSort Iterative")
plt.plot(cases,times3, ':', label="Merge Sort")
plt.plot(cases,times4, '-', label="Bubble Sort")
plt.title('Comparison')
plt.xlabel('Array Size')
plt.ylabel('Time (seconds)')
plt.legend()
plt.show()
```


Comparison (No Bubble Sort)
Heap Sort
QuickSort
QuickSort Iterative
Merge Sort
Time (seconds)
0
2
4
6
8
0
20000
40000
60000
80000
100000
Array Size



Time/Size	10	100	1000	10000	100000
Heap Sort	0.0	0.000804901123046875	0.0062978267669677734	0.0608668327331543	0.7148110866546631
QuickSort	0.0011339187622070312	0.0	0.0020549297333276367	0.06781744956970215	4.936222553253174
QuickSort Iterative	0.0	0.0	0.0019943714141845703	0.09175348281860352	8.234138011932373
Merge Sort	0.0	0.0	0.0029916763305664062	0.037412166595458984	1.1898260116577148
Bubble Sort	0.0	0.0009970664978027344	0.0907900333404541	8.598121643066406	853.2158970832825

CONCLUSION

Heap Sort, Merge Sort, QuickSort, Iterative QuickSort, and Bubble Sort are all comparison-based sorting algorithms, which means they compare elements of the input list to sort them. However, they differ in their efficiency and implementation.

1. **Bubble Sort** is an intuitive and simple sorting algorithm that repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order. It has an average and worst-case time complexity of O(n^2), making it suitable only for small datasets.

2. **Heap Sort** is an efficient sorting algorithm that uses a binary heap data structure to sort elements. It has a worst-case time complexity of O(n log n), making it suitable for large datasets. However, it has a high constant factor and requires additional space to store the heap.

3. **Merge Sort** is a divide-and-conquer sorting algorithm that divides the input array into two halves, recursively sorts each half, and merges the sorted halves. It has a worst-case time complexity of O(n log n) and requires additional space to merge the subarrays.

4. **QuickSort** is also a divide-and-conquer algorithm that selects a pivot element and partitions the input array around the pivot. It has a worst-case time complexity of O(n^2), but its average case performance is O(n log n). It is generally faster than Merge Sort and does not require additional space to merge the subarrays.

5. **Iterative QuickSort** is an implementation of QuickSort that uses a stack instead of recursion. It has the same time complexity as QuickSort and does not suffer from the recursion overhead, making it more efficient for large datasets.

In summary, Heap Sort and Merge Sort have a guaranteed worst-case time complexity of O(n log n) and are suitable for large datasets. QuickSort and Iterative QuickSort have an average case time complexity of O(n log n) and are faster than Merge Sort and Heap Sort in practice. Bubble Sort is suitable only for small datasets and is less efficient than the other algorithms.