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

Elaborated:
st. gr. FAF-213 Bardier Andrei

Verified:
asist. univ. Fiștic Cristofor

Chişinău - 2023

TABLE OF CONTENTS

ALGORITHM ANALYSIS

Objective

Study and analyze different sorting algorithms, including merge sort, quick sort, heap sort and one for my choice.

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:

An alternative to mathematical analysis of complexity is empirical analysis, which can provide useful information about the efficiency of algorithms. The stages of empirical analysis typically include:

1. Establishing the purpose of the analysis.
2. Choosing an efficiency metric, such as the number of operations executed or the execution time of the algorithm or a portion of it.
3. Defining the properties of the input data that the analysis will be based on, such as size or specific properties.
4. Implementing the algorithm in a programming language.
5. Generating multiple sets of input data.
6. Running the program for each set of input data.
7. Analyzing the obtained data.

The choice of the efficiency measure depends on the purpose of the analysis. For example, if the goal is to obtain information about the complexity class or verify a theoretical estimate, the number of operations performed is appropriate, while the execution time is more relevant if the aim is to evaluate the implementation of the algorithm.

It is important to use multiple sets of input data to reflect the different characteristics of the algorithm, covering a range of sizes and different values or configurations of input data. To obtain meaningful results, it is crucial to use input data that reflects the range of situations the algorithm will encounter in practice.

To perform empirical analysis of an algorithm's implementation in a programming language, monitoring sequences are introduced to measure the efficiency metric. If the efficiency metric is the number of operations executed, a counter is used to increment after each execution. If the metric is execution time, the time of entry and exit must be recorded. Most programming languages have functions to measure the time elapsed between two moments, and it is important to only count the time affected by the execution of the analyzed program, especially when measuring time.

After the program has been run for the test data, the results are recorded and either synthetic quantities, such as mean and standard deviation, are calculated or the results are represented graphically as pairs of points in the form of (problem size, efficiency measure).

Introduction:

Sorting algorithms are essential in programming, and they have been in use for centuries. Sorting is a process of arranging data or information in a specific order, either in ascending or descending. Sorting algorithms are used in various applications like data processing, searching, and decision-making. There are many sorting algorithms available, and the choice of algorithm depends on the size of the dataset, the distribution of the data, and the available memory.

The history of sorting algorithms dates back to ancient times when people used to sort goods in markets. In the early 1940s, computer scientists started working on sorting algorithms for computers. One of the first sorting algorithms was Bubble Sort, which was introduced in 1956. It is a simple sorting algorithm that is easy to understand and implement, but it has poor time complexity and is not suitable for large datasets.

In the 1960s, Quick Sort and Merge Sort were introduced. Quick Sort is a fast algorithm with good time complexity, but it is less stable than other sorting algorithms. Merge Sort, on the other hand, is a stable sorting algorithm that has a better time complexity than Bubble Sort and Quick Sort.

Heap Sort was introduced in the 1970s and is based on the Heap data structure. Heap Sort has an efficient time complexity and is suitable for large datasets.

In summary, sorting algorithms are crucial in programming, and they have evolved over time. The choice of sorting algorithm depends on various factors like the size of the dataset, the distribution of the data, and the available memory. Understanding the characteristics of different sorting algorithms can help programmers choose the most appropriate one for their needs.

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 five series of numbers:

- Small array of 100 random elements
- Medium array of 10’000 random elements
- Large array of 1’000’000 random elements
- Large sorted array
- Large partially sorted array (1 : 1 sorted and unsorted parts)

IMPLEMENTATION

The five algorithms will be implemented without optimization in Python and evaluated based on their completion time. The results obtained may resemble those of other similar experiments, but the specific performance in relation to the input size may vary depending on the memory specifications of the device being used.

Bubble Sort Algorithm

Bubble Sort is a simple sorting algorithm that is used to sort arrays of elements in ascending or descending order. It is one of the most basic sorting algorithms and is often used in educational contexts to teach programming students about sorting algorithms. The algorithm works by repeatedly swapping adjacent elements if they are in the wrong order until the array is sorted.

The bubble sort algorithm gets its name from the way that elements "bubble up" to their correct positions in the sorted array. In each pass through the array, the algorithm compares adjacent elements and swaps them if they are in the wrong order. This process is repeated until the array is fully sorted.

While bubble sort is easy to understand and implement, it is not very efficient for large arrays. Its worst-case time complexity is O(n^2), which means that its performance degrades quickly as the size of the input array grows.

However, bubble sort can still be useful in certain scenarios, such as when sorting small arrays or when the array is already mostly sorted. It is also sometimes used as a building block for more complex sorting algorithms.

Here's the implementation of the bubble sort algorithm in Python:

```
def bubble_sort(arr):
    n = len(arr)
    # Traverse through all array elements
    for i in range(n - 1):
        # optimization
        swapped = False
        for j in range(0, n - i - 1):
            # Swap if the element found is greater than the next element
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True
        # if no swaps happened => array already sorted
        #if not swapped:
         #   break
    return arr
```

Merge Sort

Merge Sort is a popular divide and conquer sorting algorithm that works by recursively dividing the input array into smaller subarrays until the subarrays can be easily sorted. Then, the subarrays are merged back together, with their elements being compared and arranged in order. This process is repeated until the entire array is sorted.

The basic idea behind Merge Sort is to repeatedly divide the input array into halves until each half contains only one element. Then, the subarrays are merged back together by comparing and combining the elements in a way that results in a sorted array.

One of the key benefits of Merge Sort is its efficient worst-case time complexity of O(n*log n). This makes it much faster than algorithms like Bubble Sort or Selection Sort, especially for larger arrays. Merge Sort is also stable, meaning that elements with equal values will maintain their relative order in the sorted array. This is an important feature in some applications where the order of equal elements matters.

However, Merge Sort does require additional memory for the temporary arrays used during the merging process. This can be a limitation in applications where memory usage is a concern.

Overall, Merge Sort is a powerful and efficient sorting algorithm that is widely used in a variety of applications, including database management, data compression, and numerical analysis.

Here's the implementation of the merge sort algorithm in Python:

```
def merge_sort(arr):
    # Base case: if the array has only one element, it is already sorted
    if len(arr) <= 1:
        return arr

    # Recursive case: divide the array into two halves and sort each half
    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]
    left_half = merge_sort(left_half)
    right_half = merge_sort(right_half)

    # Merge the sorted halves
    i = j = k = 0
    while i < len(left_half) and j < len(right_half):
        if left_half[i] < right_half[j]:
            arr[k] = left_half[i]
            i += 1
        else:
            arr[k] = right_half[j]
            j += 1
        k += 1
    while i < len(left_half):
        arr[k] = left_half[i]
        i += 1
        k += 1
    while j < len(right_half):
        arr[k] = right_half[j]
        j += 1
        k += 1

    return arr
```

Heap Sort

Heap Sort is a comparison-based sorting algorithm that uses a heap data structure to sort elements in an array. It works by creating a binary heap out of the input array and repeatedly removing the maximum (or minimum) element from the heap and placing it at the end of the array. The heap is then reconstructed and the process is repeated until the entire array is sorted.

Heap Sort has a time complexity of O(n*log n) in the worst case, which makes it more efficient than some other sorting algorithms like Bubble Sort or Insertion Sort. Additionally, unlike Merge Sort, Heap Sort does not require any additional memory allocation beyond the original array.

One drawback of Heap Sort is that it is not stable, meaning that elements with equal values may not maintain their relative order in the sorted array. Additionally, the implementation of Heap Sort can be more complex than other sorting algorithms.

Overall, Heap Sort is a powerful and efficient sorting algorithm that is widely used in applications such as operating system memory management and priority queue implementation.

Here's the implementation of the heap sort algorithm in Python:

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

    # If left child is larger than root
    if left < n and arr[i] < arr[left]:
        largest = left

    # If right child is larger than largest so far
    if right < n and arr[largest] < arr[right]:
        largest = right

    # If largest is not root
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]  # swap

        # Recursively heapify the affected sub-tree
        heapify(arr, n, largest)
def heap_sort(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)
```

Quick Sort

Quick Sort is a popular divide and conquer sorting algorithm that works by partitioning the input array into two subarrays around a pivot element. The pivot element is chosen from the array, and the elements in the array are rearranged such that all elements smaller than the pivot are to its left, and all elements greater than the pivot are to its right. The process is then repeated recursively on the two subarrays until the entire array is sorted.

Quick Sort has a worst-case time complexity of O(n^2), but in practice it is often much faster than other sorting algorithms with similar worst-case time complexity, such as Insertion Sort or Selection Sort. When implemented properly, Quick Sort has an average case time complexity of O(n*log n), which makes it one of the fastest sorting algorithms.

One of the advantages of Quick Sort is that it is an in-place sorting algorithm, meaning that it doesn't require any additional memory allocation beyond the original array. This makes it a good choice for sorting large arrays where memory usage is a concern.

However, Quick Sort is not stable, meaning that elements with equal values may not maintain their relative order in the sorted array. Additionally, the choice of pivot element can significantly affect the performance of the algorithm. A bad choice of pivot can lead to poor performance, particularly in the worst case.

Overall, Quick Sort is a powerful and widely-used sorting algorithm that can be very efficient when implemented properly.

Here's the implementation of the quick sort algorithm in Python:

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr  # base case

    pivot = arr[len(arr) // 2]  # choose pivot element
    left = [x for x in arr if x < pivot]  # elements less than pivot
    middle = [x for x in arr if x == pivot]  # elements equal to pivot
    right = [x for x in arr if x > pivot]  # elements greater than pivot

    # recursively sort the left and right subarrays
    return quick_sort(left) + middle + quick_sort(right)
```

Results

Bubble sort, merge sort, heap sort, and quick sort are all popular sorting algorithms used to arrange elements in an array in ascending or descending order. However, they differ in their efficiency, stability, and memory usage.

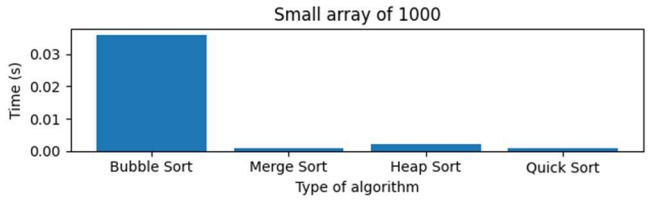
Bubble sort is a simple sorting algorithm that repeatedly compares adjacent elements and swaps them if they are in the wrong order. While it is easy to understand and implement, it is not very efficient for large arrays.

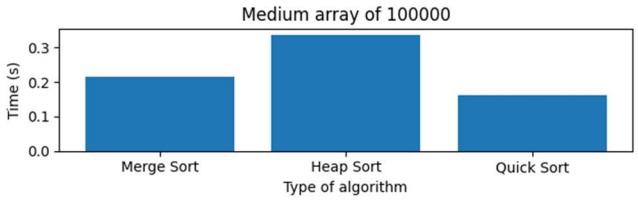
Merge sort and heap sort are both divide-and-conquer sorting algorithms that have a time complexity of O(n*log n) in the worst case. Merge sort is stable, while heap sort is not. Heap sort has the advantage of being an in-place sorting algorithm, meaning it does not require additional memory allocation beyond the original array.

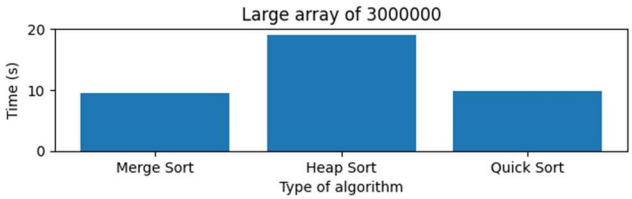
Quick sort is another divide-and-conquer sorting algorithm that has an average case time complexity of O(n*log n). It is also an in-place sorting algorithm, but is not stable. The choice of pivot element can significantly affect its performance.

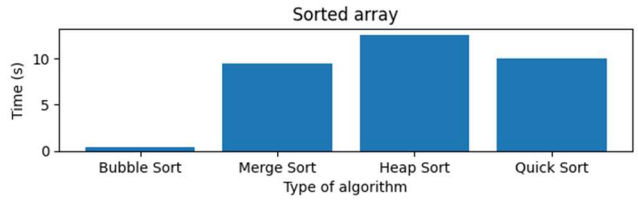
Overall, the choice of sorting algorithm depends on the specific requirements of the application, such as the size of the array, the desired stability of the sort, and memory constraints.

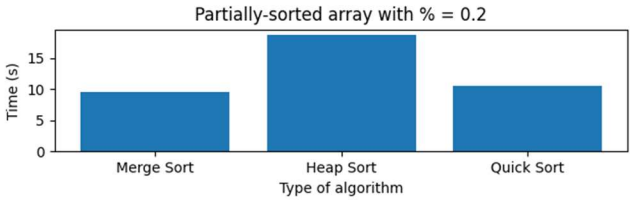
First analysis:

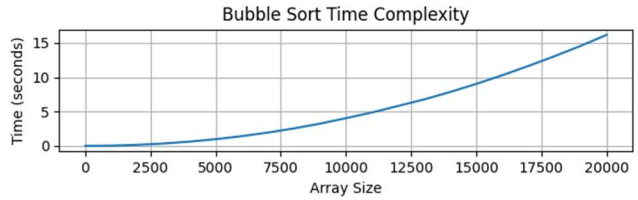




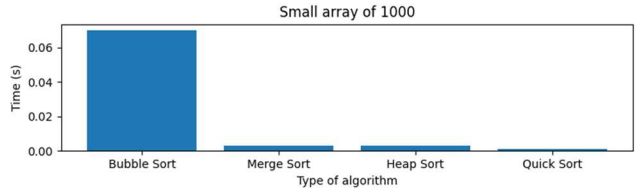


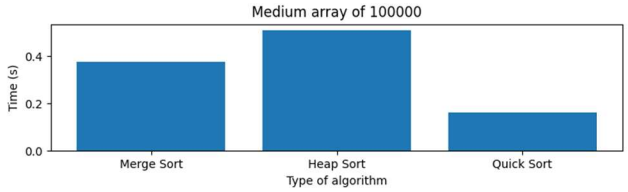


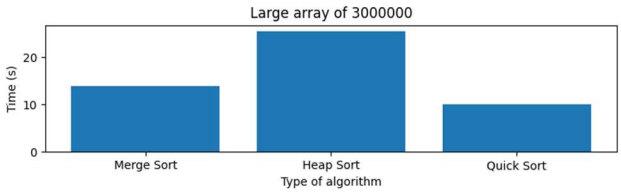


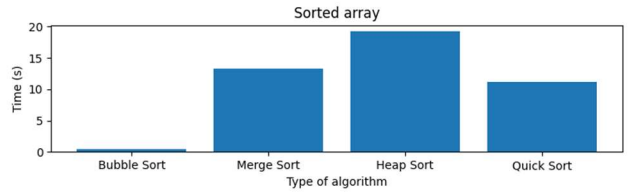


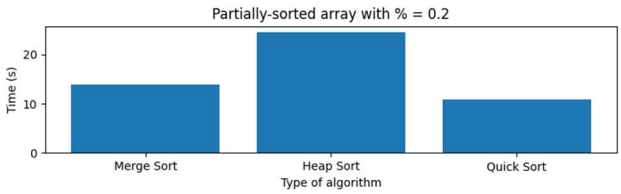
Second analysis (another PC):

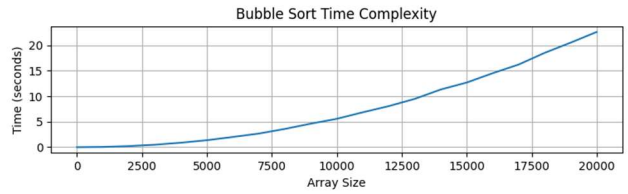












BONUS. Sleep Sort and Bogosort algorithms

Sleep sort is an unusual sorting algorithm that relies on threads and sleep functions to sort a list of integers. Although not a practical sorting algorithm for large inputs, sleep sort is an interesting programming exercise that can be implemented in Python using the threading module.

The basic idea behind sleep sort is to create a separate thread for each value to be sorted. Each thread sleeps for an amount of time proportional to the value it represents before printing out the value. Since the threads are executed in parallel, the printed values are in ascending order.

To implement sleep sort in Python, we first define a function that takes a list of integers as input. Then, we create a nested function that sleeps for the given number of seconds before appending the number to a sorted array. We create a thread for each number in the input list and start the threads. Finally, we wait for all the threads to complete before returning the sorted array.

Although the time complexity of sleep sort is O(n), the actual time taken by the algorithm depends on the maximum value in the input list. Therefore, sleep sort is mainly used as a fun and interesting programming exercise rather than a practical sorting algorithm.

```
def sleep_sort(numbers):
    sorted_arr = []
    def print_number(number):
        time.sleep(number)
        sorted_arr.append(number)

    threads = []
    for number in numbers:
        thread = threading.Thread(target=print_number, args=(number,))
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()
    return sorted_arr
```

Bogosort algorithm (Random sort)

Bogosort is a notoriously inefficient sorting algorithm that generates a random permutation of the input list and checks if it is sorted. If it is not sorted, it repeats the process until it eventually generates a sorted list. Note that bogosort has a worst-case time complexity of O(n * n!), which makes it highly impractical for even moderately sized lists. It is only included here as a curiosity and should not be used in real-world applications. Here is an implementation of bogosort in Python:

```
import random

def bogosort(lst):
    while not is_sorted(lst):
        random.shuffle(lst)
    return lst

def is_sorted(lst):
    return all(lst[i] <= lst[i+1] for i in range(len(lst)-1))
```

CONCLUSION

Sorting algorithms are an essential tool in computer science and programming. There are various sorting algorithms, each with its strengths and weaknesses. In this text, we explored four common sorting algorithms: Bubble, Merge, Heap, and Quick, and their best use cases.

Bubble sort is the simplest sorting algorithm, but it is also the slowest. It is best suited for small lists or lists that are almost sorted. Bubble sort is an in-place sorting algorithm, meaning it sorts the array by swapping adjacent elements that are out of order. However, it has a time complexity of O(n^2), which makes it impractical for large datasets.

Merge sort, on the other hand, is a divide-and-conquer algorithm that uses recursion to sort the elements. It has a time complexity of O(n log n), which makes it more efficient than Bubble sort. Merge sort is best suited for large datasets that do not fit in memory because it is a stable sorting algorithm that can handle data in external storage.

Heap sort is also a divide-and-conquer algorithm that uses a binary heap data structure to sort the elements. It has a time complexity of O(n log n) and is an in-place sorting algorithm. Heap sort is best suited for situations where the memory is limited, and there is a need for an in-place sorting algorithm.

Quick sort is another divide-and-conquer algorithm that has an average time complexity of O(n log n). It uses a pivot element to partition the array into two smaller sub-arrays, and then recursively sorts the sub-arrays. Quick sort is best suited for large datasets and is often used in practice because of its speed and efficiency.

In conclusion, the choice of sorting algorithm depends on the specific use case and the size of the dataset. Bubble sort is best suited for small datasets, while Merge sort is ideal for large datasets that do not fit in memory. Heap sort is best suited for situations where memory is limited, and Quick sort is suitable for large datasets that require fast sorting. Knowing the strengths and weaknesses of these sorting algorithms can help programmers choose the right algorithm for their specific use case.

Appendix:

GitHub repo:
https://github.com/Ricigeroi/AA-Lab-2