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, stalinSort

Elaborated:
st. gr. FAF-222 Martiniuc Artiom

Verified:
asist. univ. Fiștic Cristofor

Chişinău - 2024

TABLE OF CONTENTS

Objective

ALGORITHM ANALYSIS

Study and analyze different sorting algorithms.

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.

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:

In this laboratory exercise, we explore and analyze four different sorting algorithms: QuickSort, MergeSort, HeapSort, and StalinSort. Sorting is a fundamental operation in computer science, used in a wide array of applications from data processing to algorithm optimization. By implementing and empirically analyzing these algorithms, we aim to understand their performance characteristics in terms of time complexity, stability, and space complexity under various input conditions. Understanding the behavior, efficiency, and limitations of different sorting algorithms is crucial for selecting the appropriate method for a given dataset or application.

As mentioned previously, the performance of an algorithm can be analyzed mathematically (derived through mathematical reasoning) or empirically (based on experimental observations).

Within this laboratory, we will be analyzing the 4 naïve algorithms empirically.

Comparison Metric:

The comparison metric for this laboratory work will be considered the time of execution of each algorithm (T(n)), Amount of memory used, whether equal elements retain their original order.

Input Format:

As input, we'll use randomly generated lists of integers for our analysis.

Properties of list:

Size: Varying from 100 to 10,000 elements.

Distribution: Uniform distribution.

IMPLEMENTATION

All four algorithms will be implemented in their native form in python an analyzed empirically based on the time required for their completion. While the general trend of the results may be similar to other experimental observations, the particular efficiency in rapport with input will vary depending o memory of the device used.

The error margin determined will constitute 2.5 seconds as per experimental measurement.

Quick Sort

QuickSort is a divide-and-conquer algorithm that selects a 'pivot' element from the array and partitions the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively. This process repeats until the entire array is sorted.

Time Complexity: Average case is O(nlogn), but the worst case is $O(n^2)$ when the pivot selection is poor.

Space Complexity: O(logn) due to recursive stack space.

Stability: Not stable.

Implementation:

```
def quickSort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    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)
```

Figure 1 Quick Sort in Python

Merge Sort

MergeSort is another divide-and-conquer algorithm that divides the array into two halves, sorts each half, and then merges the two sorted halves. It guarantees stable sorting and performs well on linked lists and arrays.

Time Complexity: O(nlogn) in all cases.

Space Complexity: O(n), as it requires additional space for merging.

Stability: Stable.

Implementation:

```
def mergeSort(arr):
    if len(arr) <= 1:
        return arr
    middle = len(arr) // 2
    left = mergeSort(arr[:middle])
    right = mergeSort(arr[middle:])
    return merge(left, right)
```

Figure 2 Merge Sort in Python

```
def merge(left, right):
    result = []
    left_index, right_index = 0, 0
    while left_index < len(left) and right_index < len(right):
        if left[left_index] < right[right_index]:
            result.append(left[left_index])
            left_index += 1
        else:
            result.append(right[right_index])
            right_index += 1
    result.extend(left[left_index:])
    result.extend(right[right_index:])
    return result
```

Figure 3 Merge function in Python

Heap Sort

HeapSort utilizes a binary heap data structure to sort elements. It builds a max heap from the input data, then repeatedly removes the maximum element from the heap and restores the heap properties until the heap is empty.

Time Complexity: O(nlogn) in all cases.

Space Complexity: O(1) for in-place sorting.

Stability: Not stable.

Implementation:

```
def heapSort(arr):
    n = len(arr)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)
```

Figure 4 Heap Sort in Python

```
def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1
    r = 2 * i + 2
    if l < n and arr[i] < arr[l]:
        largest = l
    if r < n and arr[largest] < arr[r]:
        largest = r
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)
```

Figure 5 Heapify function in Python

Stalin Sort

StalinSort is a non-traditional and humoristic "sorting" algorithm where elements that are not in order are simply removed from the sequence, rather than being sorted. It's more of a joke than an actual sorting algorithm.

Time Complexity: O(n) at it only requires a single pass through the data.

Space Complexity: O(1) if elements are removed in place, or O(n) for a new array.

Stability: Not applicable as it doesnt actually sort the input.

Implementation:

```python
def stalinSort(arr):
    result = [arr[0]]
    for i in range(1, len(arr)):
        if arr[i] >= result[-1]:
            result.append(arr[i])
    return result
```

Figure 6 Stalin Sort in Python

RESULTS

The graph below illustrates the performance of each sorting algorithm (QuickSort, MergeSort, HeapSort, and StalinSort) across different array sizes ranging from 100 to 10,000 elements.

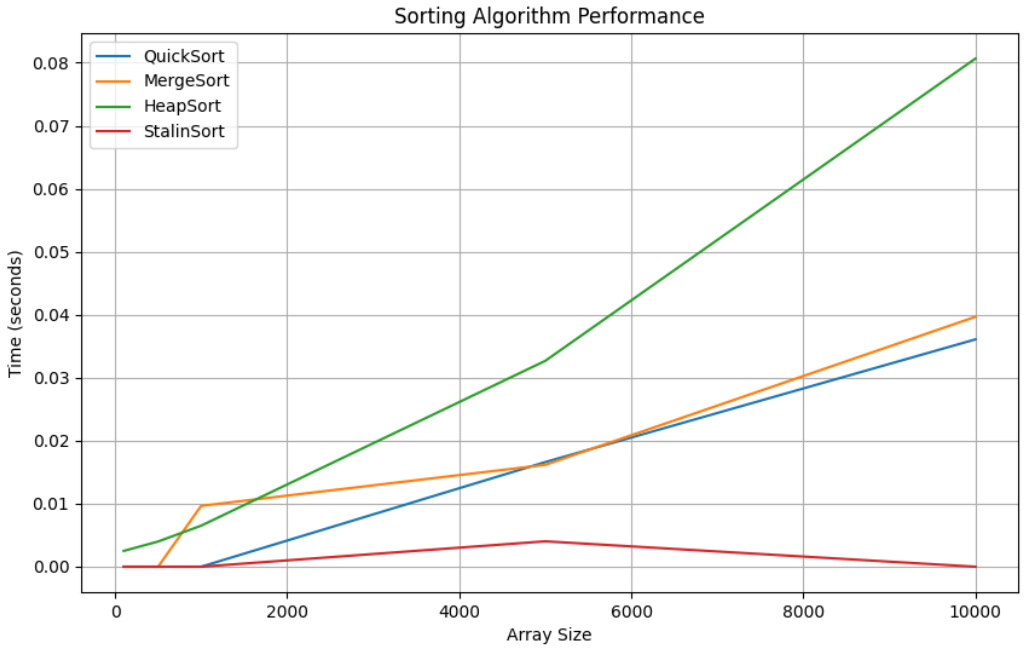

Figure 7 Graph of Sorting Algorithm Performance

QuickSort, MergeSort, and HeapSort show a gradual increase in execution time as the size of the array increases, which aligns with their expected time complexity behaviors. QuickSort and MergeSort are generally faster for smaller arrays but their time cost increases more significantly as the array size grows. HeapSort demonstrates a relatively steady increase in execution time, indicative of its consistent performance across different array sizes.

StalinSort, as expected, shows a markedly different performance characteristic. It appears to be much faster compared to the others, especially as array size increases. This is due to its simplistic approach of "eliminating" elements that are not in order, rather than performing complex sorting operations. However, it's important to note that StalinSort does not actually sort the array in the traditional sense but rather returns a subset of the original array where only the elements in a non-decreasing order are retained.

CONCLUSION

Based on this laboratory work, we can conclude that the choice of sorting algorithm can significantly impact performance and should be based on the specific requirements of the application, including the size of the dataset to be sorted and the importance of maintaining the original order of equal elements (stability).

QuickSort and MergeSort are efficient for smaller to medium-sized arrays and offer stability, making them suitable for applications where these factors are important.

HeapSort provides a good balance between time complexity and consistency, making it suitable for applications requiring predictable performance.

StalinSort, while not a practical sorting solution for traditional sorting needs, highlights the importance of algorithm choice based on the desired outcome of the sorting process.