**Ministry of Education and Research of the Republic of Moldova**

**Technical University of Moldova**

**Faculty of Computers, Informatics and Microelectronics**

**REPORT**

Laboratory work no. 2

*to Analysis Sorting Algorithms*

Did:

St. gr. TI-21X Andrei Ceban

Checked:

asist. univ. Cristofor Fistic

Chişinău - 2022

**Laboratory work no. 2**

**Objective:** Study and analyze different algorithms for determining Fibonacci n-th term

**INTRODUCTION**

Sorting algorithms are a category of algorithms that are used to arrange a collection of data in a specific order. The most common order is ascending or descending, but other orders may be used, depending on the application. Sorting algorithms are an essential tool for data processing and analysis, as they enable efficient searching, filtering, and statistical analysis of large datasets.

There are many different types of sorting algorithms, each with its own advantages and disadvantages. Some of the most popular sorting algorithms include:

Selection sort: an algorithm that finds the minimum element in the data set and swaps it with the first element, then finds the second-smallest element and swaps it with the second element, and so on.

Insertion sort: an algorithm that iteratively builds a sorted sub-array by inserting each element in its correct position.

Merge sort: a divide-and-conquer algorithm that recursively splits the data set into smaller sub-arrays, sorts them, and then merges them back together.

Quick sort: another divide-and-conquer algorithm that selects a pivot element, partitions the data set into smaller sub-arrays based on the pivot, sorts the sub-arrays recursively, and then combines them.

Each sorting algorithm has its own performance characteristics in terms of time and memory complexity, stability, and suitability for different types of data. Choosing the right sorting algorithm depends on the size and structure of the data set, the desired order, and the performance requirements of the application.

**IMPLEMENTATION**

**QuickSort**:

Quick sort is a comparison-based sorting algorithm that uses a divide-and-conquer approach to sort elements in an array. It was developed by Tony Hoare in 1959 and is one of the most commonly used sorting algorithms in practice.

Quick sort works by selecting a pivot element from the array and partitioning the array into two subarrays: one containing elements less than the pivot, and one containing elements greater than or equal to the pivot. The pivot is then placed in its final position in the sorted array, and the process is repeated recursively on the two subarrays until the entire array is sorted.

The main advantage of quick sort over other sorting algorithms is that it has an average-case time complexity of O(n log n), which is faster than many other popular sorting algorithms. However, the worst-case time complexity of quick sort is O(n^2), which can occur when the pivot is chosen poorly and the array is already partially sorted.

To mitigate this worst-case scenario, various optimizations can be applied to quick sort, such as using a randomized pivot selection or switching to a different sorting algorithm for small subarrays. Additionally, quick sort is an in-place sorting algorithm, meaning that it does not require extra space beyond the input array, which can be a benefit for memory-constrained applications.

**Code :**

def quickSort(arr):  
 if len(arr) <= 1:  
 return arr  
  
 pivot = arr[0]  
 left = [x for x in arr[1:] if x < pivot]  
 right = [x for x in arr[1:] if x >= pivot]  
  
 return quickSort(left) + [pivot] + quickSort(right)

**HeapSort**:

Heap sort is a comparison-based sorting algorithm that uses a binary heap data structure to sort elements in an array. It was first proposed by J. W. J. Williams in 1964, but was later refined and popularized by Robert Floyd in 1969.

Heap sort works by first building a heap from the input array, and then repeatedly extracting the maximum (for a max heap) or minimum (for a min heap) element from the heap and placing it at the end of the sorted portion of the array. This process is repeated until all elements are sorted.

The main advantage of heap sort over other sorting algorithms is that it has a worst-case time complexity of O(n log n), where n is the number of elements in the array, which is the same as the best-case and average-case time complexity. This makes heap sort very efficient for large datasets.

**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  
  
 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]) # swap  
 heapify(arr, n, largest)  
  
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]) # swap  
 heapify(arr, i, 0)

However, heap sort does require extra space to build the heap, which can be a disadvantage for very large datasets. Additionally, it is not a stable sorting algorithm, meaning that the order of equal elements in the input array may not be preserved in the sorted output.

**InsertionSort**:

Insertion sort is a simple sorting algorithm that works by iteratively building a sorted portion of an array. It was originally developed in the 1950s by John von Neumann, and is commonly used for small arrays and as a building block for more complex sorting algorithms.

Insertion sort works by iterating over the array from left to right, and for each element, inserting it into its correct position in the sorted portion of the array to the left. To do this, it compares the current element with each element to its left until it finds its correct position, and then shifts all the larger elements to the right by one position to make room for the new element.

The main advantage of insertion sort is its simplicity and ease of implementation, making it a good choice for small arrays or as a building block for more complex sorting algorithms. Additionally, it is an in-place sorting algorithm, meaning that it does not require extra space beyond the input array.

However, insertion sort has a worst-case time complexity of O(n^2), where n is the number of elements in the array, which makes it inefficient for larger arrays. To mitigate this, various optimizations can be applied to insertion sort, such as using binary search to find the correct position for the current element, or switching to a more efficient sorting algorithm for larger subarrays.

**Code :**

def insertionSort(arr):  
 # Traverse through 1 to len(arr)  
 for i in range(1, len(arr)):  
 key = arr[i]  
 j = i - 1  
  
 while j >= 0 and key < arr[j]:  
 arr[j + 1] = arr[j]  
 j -= 1  
 arr[j + 1] = key

**MergeSort**:

Merge sort is a comparison-based sorting algorithm that uses a divide-and-conquer approach to sort elements in an array. It was invented by John von Neumann in 1945 and is widely used in practice due to its stability, predictable performance, and ease of parallelization.

Merge sort works by recursively dividing the input array into two halves, sorting each half, and then merging the sorted halves back together. To merge the sorted halves, the algorithm compares the first element of each half and selects the smaller (or larger, depending on the desired order) element, which is then added to the output array. This process is repeated until both halves are exhausted, resulting in a fully sorted output array.

The main advantage of merge sort is its guaranteed worst-case time complexity of O(n log n), where n is the number of elements in the array, which makes it highly efficient for large datasets. Additionally, merge sort is a stable sorting algorithm, meaning that the relative order of equal elements in the input array is preserved in the sorted output.

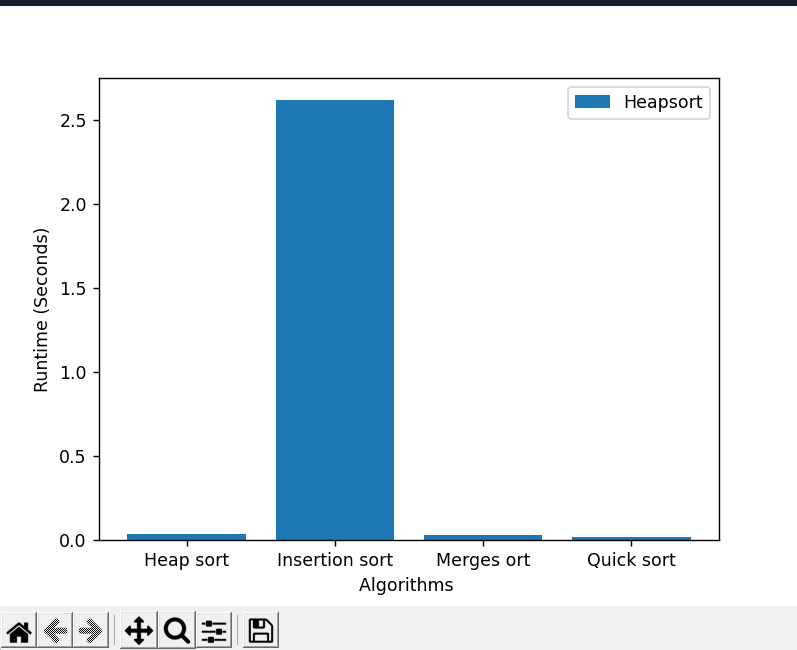
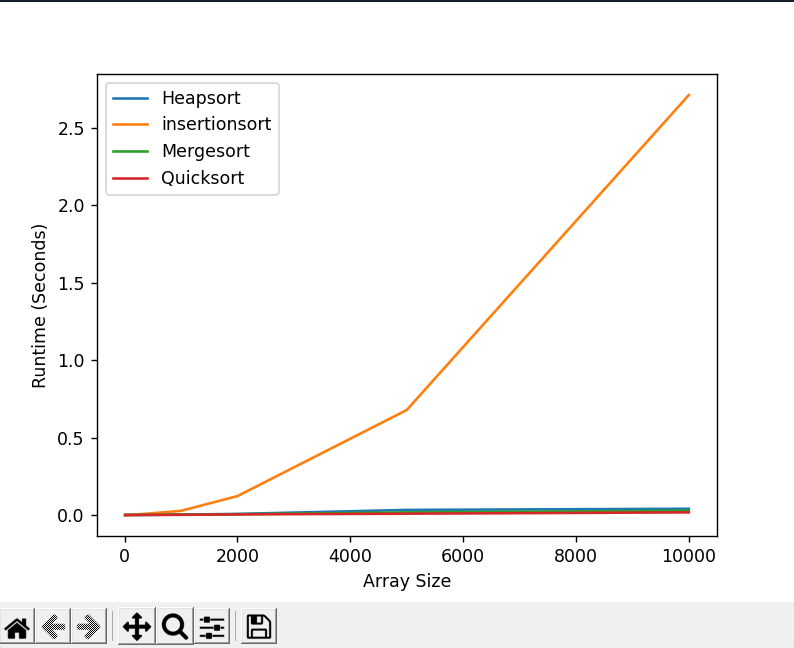
However, merge sort does require extra space to store the output array during the merge phase, which can be a disadvantage for memory-constrained applications. To mitigate this, various optimizations can be applied, such as using an in-place merge algorithm or using external memory for large datasets.

**Code :**

def mergeSort(arr):  
 if len(arr) <= 1:  
 return arr  
  
 mid = len(arr) // 2  
 left = mergeSort(arr[:mid])  
 right = mergeSort(arr[mid:])  
 result = []  
 i, j = 0, 0  
  
 while i < len(left) and j < len(right):  
 if left[i] < right[j]:  
 result.append(left[i])  
 i += 1  
 else:  
 result.append(right[j])  
 j += 1  
  
 result += left[i:]  
 result += right[j:]  
  
 return result

**RESULTS**

arr = [10, 100,1000,2000, 5000, 10000]

The time complexity of sorting algorithms can vary depending on the specific implementation and the characteristics of the data being sorted. However, in general, the time complexity of the four sorting algorithms is:

Heap sort: O(n log n) in the worst case, which is achieved when the input data is already partially sorted or has a random order.

Quick sort: O(n^2) in the worst case, but O(n log n) on average, which is achieved when the pivot element is selected randomly and the input data is not already sorted or has a random order.

Insertion sort: O(n^2) in the worst case, which is achieved when the input data is already sorted in reverse order, and O(n) in the best case, which is achieved when the input data is already sorted.

Merge sort: O(n log n) in the worst case, which is achieved regardless of the order of the input data.

**CONCLUSION**

The purpose of comparing the time complexity of heap sort, quick sort, insertion sort, and merge sort in this laboratory work is to understand the relative efficiency of these algorithms for sorting different types of data sets. By analyzing the time complexity of these algorithms, students can determine which algorithm is best suited for a given sorting problem based on the size and structure of the data set and the required performance.

Heap sort, quick sort, insertion sort, and merge sort have different time complexities and perform differently on different types of data sets. In general, merge sort has the lowest worst-case time complexity of O(n log n), making it suitable for large and complex data sets. Quick sort has an average-case time complexity of O(n log n), but can have a worst-case time complexity of O(n^2) in certain scenarios, making it less efficient for certain data sets. Heap sort has a worst-case time complexity of O(n log n), but can be less efficient than quick sort or merge sort for smaller data sets due to its higher constant factor. Insertion sort has a worst-case time complexity of O(n^2) but can be efficient for small data sets.

Git Repo : https://github.com/andeiceban0352/Labs- Anul2/tree/main/Lab%20Analiza%20Algoritmilor/Lab2