**Ministerul Educaţiei și Cercetării al Republicii Moldova**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică și Microelectronică**

**REPORT**

Laboratory Work nr.1

*at Algorithms Analysis*

Elaborated by:

st. gr. FAF-211 Hariton Dan

Verified by:

asist. univ. Fiștic Cristofor

Chişinău - 2023

**ALGORITHM ANALYSIS**

Study and analyze different algorithms for determining Fibonacci n-th term.

**Tasks:**

1. Implement at least 4 sorting algorithms

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.

**Establish Comaparation:**

We will compare these 4 algorithms using 3 arrays: one of them is 10000 terms long (10000-100000), the second one is from 100000 terms long (10000-100000), and the third one is 100000 terms long (100000-500000)

**Comparison Metric:**

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

**IMPLEMENTATION**

**QuickSort:**

import random  
import time  
start = time.time()  
  
# Function to find the partition position  
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)  
  
  
# Driver code

array = [numbers]  
print(array)  
quick\_sort(array, 0, len(array) - 1)  
  
print(f'Sorted array: {array}')  
  
print('\n')  
end = time.time()  
print(end - start)

## HeapSort:

import random  
import time  
start = time.time()  
# To heapify subtree rooted at index i.  
# n is size of heap  
  
  
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)  
  
  
# Driver's code  
if \_\_name\_\_ == '\_\_main\_\_':  
 arr = [“our numbers”]

print(arr)  
  
 # Function call  
 heapSort(arr)  
 N = len(arr)  
  
 print("Sorted array is")  
 for i in range(N):  
 print("%d" % arr[i], end=" ")  
  
print('\n')  
end = time.time()  
print(end - start)

## MergeSort:

import random  
import time  
start = time.time()  
  
# Python program for implementation of MergeSort  
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()  
  
  
# Driver Code  
if \_\_name\_\_ == '\_\_main\_\_':

array = [numbers]

printList(array)  
 mergeSort(array)  
  
  
 print("Sorted array is: ", end="\n")  
 for l in array:  
 print(l)  
  
end = time.time()  
print(end - start)

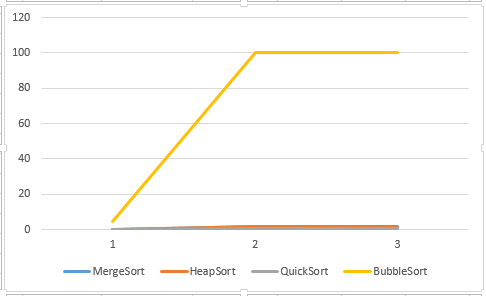
## BubbleSort:

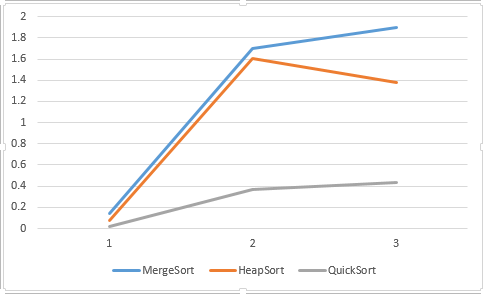
import time  
start = time.time()  
  
  
def bubbleSort(arr):  
 n = len(arr)  
  
 # Traverse through all array elements  
 for i in range(n):  
  
 # Last i elements are already in place  
 for j in range(0, n-i-1):  
  
 # traverse the array from 0 to n-i-1  
 # Swap if the element found is greater  
 # than the next element  
 if arr[j] > arr[j+1]:  
 arr[j], arr[j+1] = arr[j+1], arr[j]  
  
  
# Driver code to test above  
if \_\_name\_\_ == "\_\_main\_\_"

array = [numbers]   
  
bubbleSort(arr)  
  
print("Sorted array is:")  
for i in range(len(arr)):  
 print("%d" % arr[i], end=" ")  
  
print('\n')  
end = time.time()  
print(end - start)

**Results:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | MergeSort | HeapSort | QuickSort | BubbleSort |
| 1 | 0.14104819 | 0.07683897 | 0.024199486 | 4.698234 |
| 2 | 1.7030623 | 1.601925135 | 0.371617317 | >100 |
| 3 | 1.901052 | 1.379736185 | 0.430727959 | >100 |

****

****

**Conclusion:**

The proposed work's aim, which consists of the time-based and theoretical analysis of sorting algorithms, was successfully accomplished.Four different algorithms were used:bubbleSort, quickSort, heapSort, mergeSort to determine their temporal complexity and to highlight the most efficient algorithm that will display the desired result using the least time that passed by. The comparison of the algorithms was performed using a table where we collected time based data on the iterations received during the work and the graphs constructed in Excel, which show the noticeable difference between the complexities of the algorithms. From this we observe that the bubbleSort is the most inefficient. The rest of them look pretty even. But at a closer inspection we can see that the quickSort algorithm is the most efficient method.

**Link to GitHub**: <https://github.com/haritondan/AA-Labs>