**Ministerul Educaţiei și Cercetării al Republicii Moldova**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică și Microelectronică**

**RAPORT**

Lucrarea de laborator nr.2

*Sortarile. Tehnica Devide și Stăpânește.*

A efectuat:

st. gr. SI-211 Vozian Vladimir

A verificat:

asist. univ. Sergiu Scrob

Chişinău - 2022

**Scopul lucrării:**

De alcătuit un program ce utilizeaza trei metode de sortare: QuickSort, Merge Sort, și unul la alegere. Să se calculeze numărul de comparații și timpul pentru elementele: 10, 100, 1000, 10000. De asemenea pentru fiecare element să fie indicat cel mai bun caz, cazul mediu și cel mai rău caz.

**Codul programului:**

import random

import time

def menu():

    while True:

        print('Alegeti optiunea:\n')

        dificulty = int(input("""

        1.Cel mai favorabil caz;\n

        2.Cazul mediu;\n

        3.Cel mai defavorabil caz;\n

        0.Iesire;\n

        Alegerea:

        """))

        len\_arr=int(input("Cu cate elemente lucram? "))

        match dificulty:

            case 1:

                gen\_table = [x for x in range(1, len\_arr+1)]

                simulation(gen\_table)

            case 2:

                gen\_table = [random.randint(1, 10000) for x in range(len\_arr+1)]

                simulation(gen\_table)

            case 3:

                gen\_table = [x for x in range(len\_arr+1, 0, -1)]

                simulation(gen\_table)

            case 0:

                break

            case \_:

                print('Asa varianta nu este.')

def quick\_sort(nums):

    def \_quick\_sort(items, low, high):

        if low < high:

            split\_index = partition(items, low, high)

            \_quick\_sort(items, low, split\_index)

            \_quick\_sort(items, split\_index + 1, high)

    \_quick\_sort(nums, 0, len(nums) - 1)

def selectionSort(array, size):

    for s in range(size):

        min\_idx = s

        for i in range(s + 1, size):

            if array[i] < array[min\_idx]:

                min\_idx = i

        (array[s], array[min\_idx]) = (array[min\_idx], array[s])

    return array

def merge(left\_list, right\_list):

    sorted\_list = []

    left\_list\_index = right\_list\_index = 0

    left\_list\_length, right\_list\_length = len(left\_list), len(right\_list)

    for \_ in range(left\_list\_length + right\_list\_length):

        if left\_list\_index < left\_list\_length and right\_list\_index < right\_list\_length:

            if left\_list[left\_list\_index] <= right\_list[right\_list\_index]:

                sorted\_list.append(left\_list[left\_list\_index])

                left\_list\_index += 1

            else:

                sorted\_list.append(right\_list[right\_list\_index])

                right\_list\_index += 1

        elif left\_list\_index == left\_list\_length:

            sorted\_list.append(right\_list[right\_list\_index])

            right\_list\_index += 1

        elif right\_list\_index == right\_list\_length:

            sorted\_list.append(left\_list[left\_list\_index])

            left\_list\_index += 1

    return sorted\_list

def merge\_sort(nums):

    if len(nums) <= 1:

        return nums

    mid = len(nums) // 2

    left\_list = merge\_sort(nums[:mid])

    right\_list = merge\_sort(nums[mid:])

    return merge(left\_list, right\_list)

def insertion\_sort(list1):

    for i in range(1, len(list1)):

        a = list1[i]

        j = i - 1

        while j >= 0 and a < list1[j]:

            list1[j + 1] = list1[j]

            j -= 1

        list1[j + 1] = a

    return list1

def partition(nums, low, high):

    pivot = nums[(low + high) // 2]

    i = low - 1

    j = high + 1

    while True:

        i += 1

        while nums[i] < pivot:

            i += 1

        j -= 1

        while nums[j] > pivot:

            j -= 1

        if i >= j:

            return j

        nums[i], nums[j] = nums[j], nums[i]

def simulation(table):

    print('Tabelul introdus: \n\n', table)

    print("\n-------------------------------------------Selection sort---------------------------------------\n")

    selection\_start = time.perf\_counter\_ns()

    selection = selectionSort(table, len(table))

    selection\_end = (time.perf\_counter\_ns() - selection\_start) \* (10 \*\* -9)

    print('Tabelul sortat:\n', selection,"\n")

    print("Timpul de executie: ",selection\_end, "s\n")

    print("\n-------------------------------------------Insertion sort---------------------------------------\n")

    insertion\_start = time.perf\_counter\_ns()

    insertion = insertion\_sort(table)

    insertion\_end = (time.perf\_counter\_ns() - insertion\_start) \* (10 \*\* -9)

    print('Tabelul sortat:\n', insertion,"\n")

    print("Timpul de executie: ",insertion\_end, "s\n")

    print("\n-------------------------------------------Merge sort---------------------------------------\n")

    merge\_start = time.perf\_counter\_ns()

    merge = merge\_sort(table)

    merge\_end = (time.perf\_counter\_ns() - merge\_start) \* (10 \*\* -9)

    print('Tabelul sortat:\n', merge,"\n")

    print("Timpul de executie: ",merge\_end, "s\n")

    print("\n-------------------------------------------Quick sort---------------------------------------\n")

    quick\_start = time.perf\_counter\_ns()

    quick\_sort(table)

    quick\_end = (time.perf\_counter\_ns() - quick\_start) \* (10 \*\* -9)

    print('Tabelul sortat:\n', table,"\n")

    print("Timpul de executie: ", quick\_end, "s\n")

menu()

**Tabelul rezultatelor**

|  |  |  |  |
| --- | --- | --- | --- |
| Metoda |  | Numărul | Timpul |
| QuickSort | Cel mai bun caz | 10 | 0.0009 |
| 100 | 0.0001 |
| 1000 | 0.001 |
| 10000 | 0.04 |
| Caz mediu | 10 | 0.0005 |
| 100 | 0.0002 |
| 1000  10000 | 0.006 |
| 0.02 |
| Cel mai rău caz | 10 | 0.003 |
| 100 | 0.0003 |
| 1000 | 0.003 |
| 10000 | 0.06 |
| Merge Sort | Cel mai bun caz | 10 | 0.003 |
| 100 | 0.0002 |
| 1000 | 0.001 |
| 10000 | 0.14 |
| Caz mediu | 10 | 0.0 |
| 100 | 0.0003 |
| 1000 | 0.006 |
| 10000 | 0.04 |
| Cel mai rău caz | 10 | 0.001 |
| 100 | 0.0006 |
| 1000 | 0.008 |
| 10000 | 0.13 |
| SelectionSort | Cel mai bun caz | 10 | 0.0009 |
| 100 | 0.0004 |
| 1000 | 0.029 |
| 10000 | 2.24 |
| Caz mediu | 10 | 0.0005 |
| 100 | 0.0004 |
| 1000  10000 | 0.1 |
| 2.3 |
| Cel mai rău caz | 10 | 0.003 |
| 100 | 0.001 |
| 1000 | 0.14 |
| 10000 | 2.4 |
|  |  |  |  |
| Insertion Sort | Cel mai bun caz | 10 | 0.00001 |
| 100 | 0.0001 |
| 1000 | 0.0001 |
| 10000 | 0.001 |
| Caz mediu | 10 | 0.0 |
| 100 | 0.0001 |
| 1000 | 0.0006 |
| 10000 | 0.001 |
| Cel mai rău caz | 10 | 0.0 |
| 100 | 0.0002 |
| 1000 | 0.0006 |
| 10000 | 0.005 |

**Tabelul complexitatii**

|  |  |
| --- | --- |
| Algoritmul | Complexitatea |
| Selection sort | O(n2) |
| Insertion sort | O(n2) |
| Merge sort | O(n\*log n) |
| Quick sort | O() |

**Concluzii**

Pentru realizarea acestei lucrări de laborator, am utilizat patru metode de sortare: QuickSort, Merge Sort Selection și Insertion Sort. Am impementat algoritmele in program, am adaptat acesta pentru utilizare mai rapida ca generarea tabelului in cazurile: cel mai favorabil, mediu si cel mai nefavorabil. Dupa analiza datelor obtinute constatam ca Selection sort are cel mai mare timp de executie in toate 3 cazuri la 1000 si 10000 de elemente din array, cel mai bine a lucrat insertion sort care are cel mai mic timp de executie.