**Ministerul Educației și Cercetării al Republicii Moldova**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică și Microelectronică**



Raport

Lucrarea de laborator nr. 1

Disciplina: **Analiza si proiectarea algortimilor**

Tema: **Algoritmi de sortare**

|  |  |  |
| --- | --- | --- |
| **A efectuat:** | Student grupa TI-231 FR | Apareci Aurica |
| **A verificat:** | Asistent universitar | Andrievschi-Bagrin Veronica |

**Chișinău 2025**

**Cuprins**

[1.Cadru teoretic 3](#_Toc1)

[2. Listingul programului 4](#_Toc2)

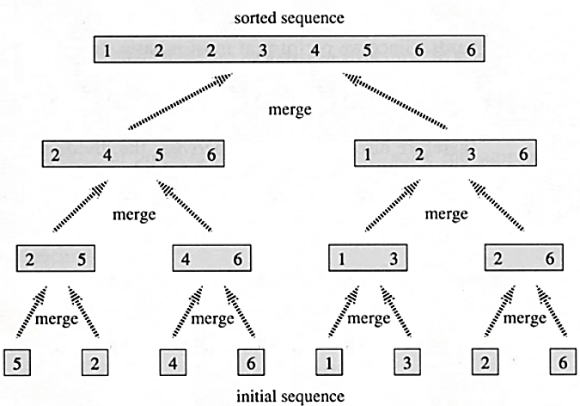
[3. Cazuri de testare 8](#_Toc3)

[4. Concluzii 12](#_Toc4)

# **Cadru teoretic**

**Tema:** Metode de sortare

**Sarcina:**Analiza si implementati algoritmii metodelor de sortare Merge Sort si Quick Sort



# **2. Listingul programului**

**//se utilizeaza urmatoarea biblioteca: GitHub - dotnet/BenchmarkDotNet: Powerful .NET library for benchmarking versiunea: 0.13.2**

**using BenchmarkDotNet.Running;**

**using BenchmarkDotNet.Attributes;**

**namespace Laboratorul\_2**

**{**

**internal class Program**

**{**

**private static void Main(string[] args)**

**{**

**BenchmarkRunner.Run<Bench>();**

**}**

**}**

**internal class SortingAlgs**

**{**

**void SwapNum(ref int x, ref int y)**

**{**

**int tempswap = x;**

**x = y;**

**y = tempswap;**

**}**

**public void bubbleSort(int[] arr)**

**{**

**int n = arr.Length;**

**for (int i = 0; i < n - 1; i++)**

**for (int j = 0; j < n - i - 1; j++)**

**if (arr[j] > arr[j + 1])**

**{**

**SwapNum(ref arr[j], ref arr[j + 1]);**

**}**

**}**

**public void bubbleSortWithFlag(int[] arr)**

**{**

**int i, j;**

**int n = arr.Length;**

**bool swapped;**

**for (i = 0; i < n - 1; i++)**

**{**

**swapped = false;**

**for (j = 0; j < n - i - 1; j++)**

**{**

**if (arr[j] > arr[j + 1])**

**{**

**SwapNum(ref arr[j], ref arr[j + 1]);**

**swapped = true;**

**}**

**}**

**if (swapped == false)**

**break;**

**}**

**}**

**void merge(int[] arr, int l, int m, int r)**

**{**

**int n1 = m - l + 1;**

**int n2 = r - m;**

**int[] L = new int[n1];**

**int[] R = new int[n2];**

**int i, j;**

**for (i = 0; i < n1; ++i)**

**L[i] = arr[l + i];**

**for (j = 0; j < n2; ++j)**

**R[j] = arr[m + 1 + j];**

**i = 0;**

**j = 0;**

**int k = l;**

**while (i < n1 && j < n2)**

**{**

**if (L[i] <= R[j])**

**{**

**arr[k] = L[i];**

**i++;**

**}**

**else**

**{**

**arr[k] = R[j];**

**j++;**

**}**

**k++;**

**}**

**while (i < n1)**

**{**

**arr[k] = L[i];**

**i++;**

**k++;**

**}**

**while (j < n2)**

**{**

**arr[k] = R[j];**

**j++;**

**k++;**

**}**

**}**

**public void MergeSort(int[] arr, int l, int r)**

**{**

**if (l < r)**

**{**

**int m = l + (r - l) / 2;**

**MergeSort(arr, l, m);**

**MergeSort(arr, m + 1, r);**

**merge(arr, l, m, r);**

**}**

**}**

**public void QuickSort(int[] arr, int low, int high)**

**{**

**if (low < high)**

**{**

**int pi = partition(arr, low, high);**

**QuickSort(arr, low, pi - 1);**

**QuickSort(arr, pi + 1, high);**

**}**

**}**

**int partition(int[] arr, int low, int high)**

**{**

**int pivot = arr[high];**

**int i = (low - 1);**

**for (int j = low; j < high; j++)**

**{**

**if (arr[j] <= pivot)**

**{**

**i++;**

**SwapNum(ref arr[i], ref arr[j]);**

**}**

**}**

**SwapNum(ref arr[i + 1], ref arr[high]);**

**return i + 1;**

**}**

**}**

**[MemoryDiagnoser]**

**public class Bench//50,100,1000,5000,10000**

**{**

**public Bench()**

**{**

**SortingAlgs = new SortingAlgs();**

**generateArray(1000, 1);**

**printInitialArray();**

**}**

**private void printInitialArray()**

**{**

**Console.WriteLine("Array-ul initial");**

**for (int i = 0; i < initialArray.Length; i++)**

**{**

**Console.Write($"{initialArray[i]} ");**

**}**

**Console.WriteLine();**

**}**

**private int[] initialArray;**

**private int[] ArrayToSort;**

**private SortingAlgs SortingAlgs { get; set; }**

**void generateArray(int count, int type)//1- crescator//2- descrescator//3- random**

**{**

**initialArray = new int[count];**

**switch (type)**

**{**

**case 1:**

**{**

**for (int i = 0; i < count; i++)**

**{**

**initialArray[i] = i;**

**}**

**initArrayToSort();**

**}**

**break;**

**case 2:**

**{**

**for (int i = 0; i < count; i++)**

**{**

**initialArray[i] = count - i;**

**}**

**initArrayToSort();**

**}**

**break;**

**case 3:**

**{**

**Random rd = new Random();**

**for (int i = 0; i < count; i++)**

**{**

**initialArray[i] = rd.Next(0,count);**

**}**

**initArrayToSort();**

**}**

**break;**

**default: Console.WriteLine("Invalid generate option"); break;**

**}**

**}**

**void initArrayToSort()**

**{**

**ArrayToSort = new int[initialArray.Length];**

**for (int i = 0; i < initialArray.Length; i++)**

**{**

**ArrayToSort[i] = initialArray[i];**

**}**

**}**

**[Benchmark]**

**public void BubleSortWithFlagBenchmark()**

**{**

**SortingAlgs.bubbleSortWithFlag(ArrayToSort);**

**initArrayToSort();**

**}**

**[Benchmark]**

**public void MergeSortBenchmark()**

**{**

**SortingAlgs.MergeSort(ArrayToSort,0,ArrayToSort.Length-1);**

**initArrayToSort();**

**}**

**[Benchmark]**

**public void QuickSortBenchmark()**

**{**

**SortingAlgs.QuickSort(ArrayToSort, 0, ArrayToSort.Length - 1);**

**initArrayToSort();**

**}**

**}**

**}**

# **3. Cazuri de testare**

N – Numărul de elemente în array | V – Versiunea de generare a array-ului

1 – array sortat crescător de N elemente

2 – array sortat descrescător de N elemente

3 – array de N elemente aleatorii

|  |
| --- |
| N = 50 , V = 1 |
|  |
| N = 50 , V = 2 |
|  |
| N = 50 , V = 3 |
|  |
| N = 100 , V = 1 |
|  |
| N = 100 , V = 2 |
|  |
| N = 100 , V = 3 |
|  |
| N = 1000 , V = 1 |
|  |
| N = 1000 , V = 2 |
|  |
| N = 1000 , V = 3 |
|  |

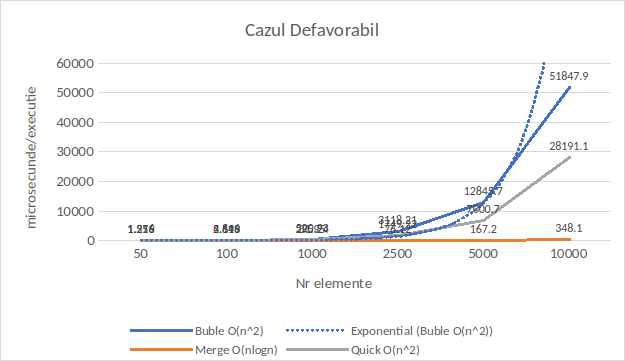
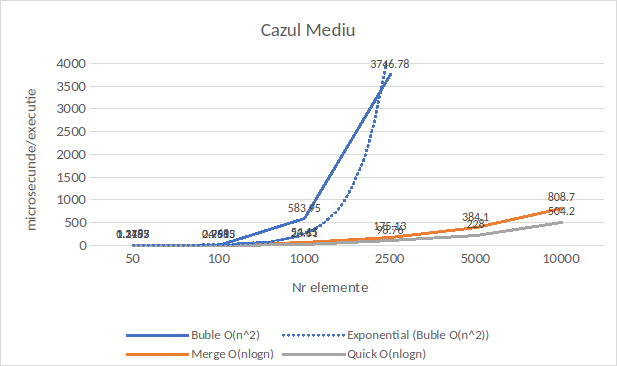
**Tabelul valorilor obtinute**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Favorabil** | | | | | | |
|  | 50 | 100 | 1000 | 2500 | 5000 | 10000 |
| Bubble O(n) | 0.0928 | 0.1663 | 1.535 | 3.618 | 7.19 | 14.14 |
| Merge O (n log n) | 1.25126 | 2.7687 | 32.097 | 82.642 | 171.657 | 360.16 |
| Quick O(n^2) | 1.65423 | 5.9555 | 418.651 | 2561.432 | 10065.445 | 40896.88 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Defavorabil** | | | | | | |
|  | 50 | 100 | 1000 | 2500 | 5000 | 10000 |
| Bubble O(n^2) | 1.556 | 5.899 | 520.93 | 3118.21 | 12845.7 | 51847.9 |
| Merge O (n log n) | 1.219 | 2.613 | 31.95 | 76.12 | 167.2 | 348.1 |
| Quick O(n^2) | 1.276 | 4.146 | 295.24 | 1749.22 | 7000.7 | 28191.1 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Mediu** | | | | | | |
|  | 50 | 100 | 1000 | 2500 | 5000 | 10000 |
| Bubble O(n^2) | 1.1493 | 4.58 | 583.95 | 3746.78 | 15231.3 | 80302.2 |
| Merge O (n log n) | 1.2187 | 2.7615 | 54.15 | 175.13 | 384.1 | 808.7 |
| Quick O (n log n) | 0.3755 | 0.9993 | 23.61 | 98.76 | 228 | 504.2 |

**Analiza valorilor obtinute**



# **4. Concluzii**

Obiectivul principal al acestei lucrări a fost de a demonstra dependenta complexității algoritmilor de sortare fața de setul de date de intrare cat si de a face comparația dintre algoritmii liniari de sortare si cei bazați pe ideea **Divide et Impera**. Drept candidați spre analiza au fost înaintați următorii algoritmi de sortare: ***Bubble Sort, Merge Sort și Quick Sort***. Pentru o analiza obiectivă am stabilit următoarele cazuri de testare:cazul favorabil (set de valori sortat deja crescător), cazul mediu (set de valori aleatoare) și cazul defavorabil (set de valori este deja sortat descrescător), unde fiecare caz de testare conține 6 seturi de elemente. Drept valori ce urmează a fi analizate (date de ieșire) vom utiliza timpul total de execuție a fiecărui algoritm pentru fiecare set de date.

În urma măsurărilor efectuate și a analizei datelor obținute putem confirma faptul că timpul de execuție al algoritmilor de sortare este dependent de setul de date de intrare si anume de natura acestuia. Drept algoritm optim de sortare din cele analizate, pot înainta Merge Sort ndatorita complexității sale de O(n log n) pentru oricare caz de testare, însă nu trebuie sa uitam de complexitatea de memorie a acestui algoritm. Un alt lucru de remarcat este viteza de execuție pentru algoritmul Quick Sorti in cazul mediu, unde arata cel mai mic timp de execuție si ne prezinta o complexitate O(n log n) însă partea negativa a acestuia apare in cazurile de testare favorabil si defavorabil, unde seturile de intrare sunt deja sortate crescător si descrescător. Pentru algoritmul Bubble Sorts-a folosit o varianta optimizata, unde este adăugat un flag de verificare ce scade complexitatea algoritmului la O(n) pentru cazul favorabil, motiv pentru care in cazul favorabil acesta prezinta cele mai buna rezultate.