**Ministerul Educaţiei și Cercetării al Republicii Moldova**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică și Microelectronică**

**RAPORT**

Lucrarea de laborator nr.2

*la Analiza și Proiectarea Algoritmilor*

A efectuat:

st. gr. SI-221 Ceclea Victor

A verificat:

asist. univ. Andrievschi-Bagrin Veronica

Chişinău - 2023

**Lucrare de laborator nr 2**

**Tema:** Analiza algoritmilor

**Scopul lucrării**:

1. Studierea metodei divide et impera.

2. Analiza şi implementarea algoritmilor bazaţi pe metoda divide et impera

**Sarcina**:

1. De efectuat analiza empirică a algoritmilor propuși

2. De determinat relația ce reprezintă complexitatea temporală pentru acești algoritmi

3. De determinat complexitatea asimptotică a algoritmilor

**Rezumat succint la tema lucrării de laborator:**

**Quick Sort:**

Quick Sort este un algoritm de sortare eficient și rapid care utilizează divizarea și stăpânirea (divide and conquer) pentru a sorta un vector. A fost inventat de Tony Hoare în 1959 și este unul dintre cele mai eficiente algoritmi de sortare în practică.

Iată pașii principali ai algoritmului Quick Sort:

Alegerea unui pivot: Se alege un element din vector, numit "pivot". Alegerea pivotului poate afecta performanța algoritmului, dar în mod obișnuit se alege primul element, ultimul element sau un element ales aleatoriu.

Partiționarea: Se rearanjează elementele din vector astfel încât toate elementele mai mici decât pivotul să fie plasate înaintea lui, iar toate elementele mai mari să fie plasate după el. Pivotul însuși se află pe poziția sa finală în vector. Acest lucru se face astfel încât pivotul să fie în poziția sa corectă în vectorul sortat.

Recursivitate: Se aplică recursiv aceeași procedură asupra subvectorilor stâng și drept creați în urma partionării. Aceasta înseamnă că se sortează subvectorii mai mici până când întregul vector este sortat.

Algoritmul Quick Sort folosește recursivitatea pentru a atinge aceste etape. Iată un exemplu:

Alegerea pivotului:

Avem un vector nesortat: [38, 27, 43, 3, 9, 82, 10]

Putem alege, de exemplu, ultimul element, care este 10, drept pivot.

Partiționarea:

Rearanjăm elementele astfel încât cele mai mici decât pivotul să fie în stânga, iar cele mai mari să fie în dreapta. După aceasta, pivotul va fi pe poziția sa corectă.

După această etapă, vectorul ar putea arăta așa: [3, 9, 10, 27, 43, 82, 38]

Recursivitate:

Aplicăm același proces de alegere a pivotului, partiționare și recursivitate pentru subvectorii stâng și drept. Aceasta se continuă până când subvectorii conțin un singur element (care este considerat sortat).

Rezultatul final este un vector sortat.

Quick Sort are o performanță medie de O(n log n) și este unul dintre cele mai rapide algoritmi de sortare în majoritatea cazurilor. Cu toate acestea, performanța sa poate scădea în cazul în care pivotul este ales în mod suboptimal (de exemplu, atunci când vectorul este deja parțial sortat).

**Merge Sort:**

Merge Sort este un algoritm de sortare eficient și stabil care utilizează divizarea și stăpânirea (divide and conquer) pentru a sorta un vector. A fost inventat de matematicianul și informaticianul John von Neumann în 1945.

Iată pașii principali ai algoritmului Merge Sort:

Divizare: Se împarte vectorul inițial în două subvectori de dimensiuni aproximativ egale. Acest proces continuă până când fiecare subvector conține un singur element, moment în care subvectorii sunt considerați sortați.

Combinație (Merge): Se combină subvectorii sortați înapoi în vectorul original. Aceasta se face astfel încât fiecare element din vector să fie la locul său corect. Procesul de combinare păstrează ordinea sortată.

Algoritmul Merge Sort folosește recursivitatea pentru a atinge aceste două etape. Iată un exemplu:

Pasul de divizare:

Avem un vector nesortat: [38, 27, 43, 3, 9, 82, 10]

Se dividează în două subvectori: [38, 27, 43] și [3, 9, 82, 10]

Sortarea subvectorilor:

Se continuă divizarea până când fiecare subvector are un singur element. Acestea sunt considerate sortate în sine.

Pasul de combinare (Merge):

Se iau subvectorii sortați și se combină înapoi în vectorul original, astfel încât fiecare element să fie la locul său corect. Acest lucru se face comparând elementele și plasându-le în ordine crescătoare.

Pentru exemplul nostru: [27, 38, 43] și [3, 9, 10, 82] se combină în [3, 9, 10, 27, 38, 43, 82].

Rezultatul final este un vector sortat.

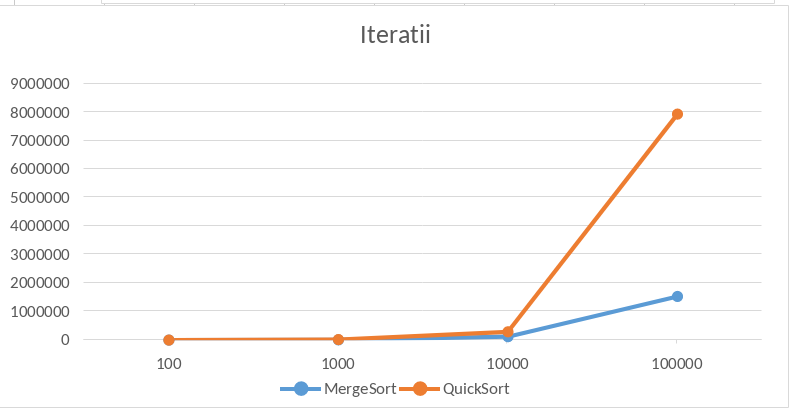
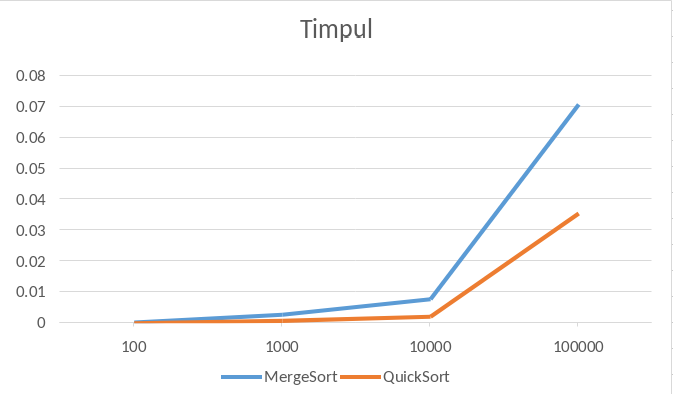
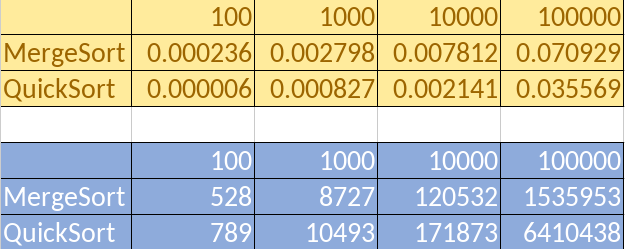
Unul dintre avantajele Merge Sort este că are complexitatea timp de O(n log n) în toate cazurile, ceea ce îl face eficient pentru seturi de date mari. De asemenea, Merge Sort este stabil, ceea ce înseamnă că păstrează ordinea relativă a elementelor egale.

Cu toate acestea, Merge Sort utilizează spațiu suplimentar în comparație cu alți algoritmi de sortare (cum ar fi Quick Sort), deoarece necesită un spațiu auxiliar pentru a combina subvectorii.

**Codul programului in JS**

#include <iostream>  
#include <vector>  
#include <random>  
#include <chrono>  
int merge(std::vector<int>& arr, int left, int middle, int right) {  
 int n1 = middle - left + 1;  
 int n2 = right - middle;  
  
 std::vector<int> leftArr(n1);  
 std::vector<int> rightArr(n2);  
  
 for (int i = 0; i < n1; i++) {  
 leftArr[i] = arr[left + i];  
 }  
  
 for (int i = 0; i < n2; i++) {  
 rightArr[i] = arr[middle + 1 + i];  
 }  
  
 int i = 0, j = 0, k = left;  
 int comparisons = 0;  
  
 while (i < n1 && j < n2) {  
 comparisons++;  
 if (leftArr[i] <= rightArr[j]) {  
 arr[k++] = leftArr[i++];  
 } else {  
 arr[k++] = rightArr[j++];  
 }  
 }  
  
 while (i < n1) {  
 arr[k++] = leftArr[i++];  
 }  
  
 while (j < n2) {  
 arr[k++] = rightArr[j++];  
 }  
  
 return comparisons;  
}  
  
int mergeSort(std::vector<int>& arr, int left, int right) {  
 if (left < right) {  
 int middle = left + (right - left) / 2;  
  
 int comparisonsLeft = mergeSort(arr, left, middle);  
 int comparisonsRight = mergeSort(arr, middle + 1, right);  
  
 int comparisonsMerge = merge(arr, left, middle, right);  
  
 return comparisonsLeft + comparisonsRight + comparisonsMerge;  
 }  
  
 return 0;  
}  
  
int partition(std::vector<int>& arr, int low, int high, int& comparisons) {  
 int pivot = arr[high];  
 int i = low - 1;  
  
 for (int j = low; j <= high - 1; j++) {  
 comparisons++;  
 if (arr[j] < pivot) {  
 i++;  
 std::swap(arr[i], arr[j]);  
 }  
 }  
  
 std::swap(arr[i + 1], arr[high]);  
 return i + 1;  
}  
  
void quickSort(std::vector<int>& arr, int low, int high, int& comparisons) {  
 if (low < high) {  
 int pi = partition(arr, low, high, comparisons);  
  
 quickSort(arr, low, pi - 1, comparisons);  
 quickSort(arr, pi + 1, high, comparisons);  
 }  
}  
  
std::vector<int> generateRandomDigits(int length) {  
 std::vector<int> result;  
 std::random\_device rd;  
 std::mt19937 gen(rd());  
 std::uniform\_int\_distribution<> dis(0, 1000);  
  
 for (int i = 0; i < length; i++) {  
 result.push\_back(dis(gen));  
 }  
  
 return result;  
}  
  
int main() {  
 int lungime;  
 std::cout << "Cate elemente sa aiba Vectorul:";  
 std::cin>>lungime;  
 std::vector<int> arr = generateRandomDigits(lungime);  
 std::vector<int> arr2 = arr;  
 int n = arr.size();  
  
 auto start1 = std::chrono::high\_resolution\_clock::now();  
 int comparisons1 = mergeSort(arr, 0, n - 1);  
 auto end1 = std::chrono::high\_resolution\_clock::now();  
 std::chrono::duration<double> duration1 = end1 - start1;  
 std::cout << "Time taken by function MergeSort: "  
 << duration1.count() << " seconds" << std::endl;  
  
  
 auto start = std::chrono::high\_resolution\_clock::now();  
 int comparisons2 = 0;  
 quickSort(arr2, 0, n - 1, comparisons2);  
 auto end = std::chrono::high\_resolution\_clock::now();  
 std::chrono::duration<double> duration = end - start;  
 std::cout << "Time taken by function Quick Sort: "  
 << duration.count() << " seconds" << std::endl;  
  
  
 std::cout << "Vectorul Sortat cu MergeSort: ";  
 for (int i = 0; i < n; i++) {  
 std::cout << arr[i] << " ";  
 }  
  
 std::cout << "\n";  
  
 std::cout << "Vectorul Sortat cu QuickSort: ";  
 for (int i = 0; i < n; i++) {  
 std::cout << arr[i] << " ";  
 }  
  
 std::cout << "\nNumber of comparisons MergeSort: " << comparisons1;  
  
 std::cout << "\nNumber of comparisons QuickSort: " << comparisons2;  
  
 return 0;  
}

**Analiza empirică a algoritmilor**



**Concluzie**

În cadrul acestei lucrări de laborator, am avut ocazia să studiem și să comparăm două dintre cele mai eficiente algoritmi de sortare: Quick Sort și Merge Sort.

1. Am început prin a înțelege și a implementa algoritmul Quick Sort, care se bazează pe principiul divizării și stăpânirii, și are o complexitate medie de O(n log n).

2. A urmat studiul și implementarea algoritmului Merge Sort, care, de asemenea, utilizează divizarea și stăpânirea, oferind o performanță excelentă cu o complexitate medie de O(n log n).

3. Am observat că Quick Sort este mai eficient în majoritatea cazurilor, datorită faptului că are o constantă mai mică în factorul de complexitate, în timp ce Merge Sort necesită o cantitate mai mare de spațiu suplimentar pentru a combina subvectorii.

4. Cu toate acestea, Merge Sort este un algoritm stabil, ceea ce înseamnă că păstrează ordinea relativă a elementelor egale, în timp ce Quick Sort nu garantează această stabilitate.

5. Am analizat și modurile în care alegerea pivotului poate influența performanța algoritmului Quick Sort, fiind crucial să alegem un pivot care să minimizeze numărul de comparații.

6. În ceea ce privește Merge Sort, acesta nu depinde de alegerea pivotului, deoarece împarte vectorul în mod egal, iar combinația ulterioară asigură ordinea corectă.

7. Am experimentat cu ambele algoritmi folosind seturi de date de diferite dimensiuni și compoziții, observând că performanța lor se menține în general în timpul diverselor scenarii de testare.

8. Această lucrare de laborator a contribuit la consolidarea înțelegerii noastre asupra principiilor fundamentale ale algoritmilor de sortare și la abilitatea noastră de a alege algoritmul potrivit în funcție de cerințele specifice ale unui proiect.

9. În concluzie, am învățat că Quick Sort și Merge Sort sunt două instrumente extrem de utile în manipularea și organizarea datelor și reprezintă un punct de plecare solid în explorarea și înțelegerea altor algoritmi complexi.