MINISTERUL EDUCAŢIEI

Universitatea Tehnică a Moldovei

Facultatea Calculatoare Informatică și Microelectronică

Tehnologia Informației

Raport

Disciplina: Analiza și proiectarea algoritmilor

Lucrarea de laborator nr. 2

Tema: *“* Metoda divide et impera.*”*

Student: _____ **Raevschi Grigore TI-231**

Coordonator: _____ **Asistent univ. Coșer Cătălin**

Chișinău 2024

Cuprins

Scopul lucrării:

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

Sarcina de bază:

1. Studiați noţiunile teoretice despre metoda divide et impera
2. Implementaţi algoritmii Mergesort şi Quicksort
3. Efectuaţi analiza empirică a algoritmilor Mergesort şi Quicksort
4. Faceți o concluzie asupra lucrării.

Introducere

Metoda **„Divide et Impera”** este un procedeu recursive de rezolvare al problemelor care pot fi descompuse în subprobleme de același tip ce se rezolvă mai ușor decât problema inițială; rezultatele obținute se combină pe parcurs pentru a-l formula pe cel final.

Fiind un procedeu recursiv, subproblemele trebuie să îndeplinească următoarele caracteristici:

— să fie de același tip cu problema principală (cea care se descompune);
— să fie mai ușoare, mai simple de rezolvat;
— să nu se intercaleze cu celelalte subprobleme.

Algorimtul în pseudocod:

```
function divimp(x) {
    // returnează o soluție pentru cazul x
    if (x este suficient de mic) {
        // returnează soluția ad-hoc pentru x
        return adhoc(x);  } // descompune x în subcazurile x1, x2, …, xk
    subcazuri = descompune(x);
    for (i = 1; i <= k; i++) {
        yi = divimp(subcazuri[i]);
    } // recompune y1, y2, …, yk în scopul obținerii soluției y pentru x
    y = recompune(yi);
    return y; }
```

Quick Sort

Ca și Merge Sort, Quick Sort se bazează pe paradigma divide et impera:

1. **Divide**: Împarte array-ul A[p..r] în două sub-array-uri A[p..q-1] și A[q+1..r], unde fiecare element din A[p..q-1] este mai mic sau egal cu A[q] și fiecare element din A[q+1..r] este mai mare sau egal cu A[q];
2. **Impera**: Sortează cele două sub-array-uri aplicând recursiv algoritmul quicksort;
3. **Combina**: Deoarece cele două sub-array-uri sunt deja sortate, array-ul original este deja sortat.

Complexitatea în timp în cel mai rău caz este $\Theta(n^2)$, dar cu o distribuție liniară a elementelor, algoritmul este executat în $\Theta(n * \lg(n))$.

În cel mai nefavorabil caz, Quick Sort poate avea o complexitate pătratică, dar în medie are o performanță foarte bună și este frecvent folosit în practică.

Codul programului vezi în ANEXA A

Merge Sort

Merge Sort se bazează pe paradigma divide et impera:

— **Divide**: Problema este divizată recursiv în două subprobleme, fiecare având jumătate din complexitatea problemei inițiale;

— **Impera**: Descompunem recursiv problema până când subproblema devine banală și poate fi rezolvată într-un singur pas;

— **Combina**: Procedura MERGE combină subproblemele pentru a rezolva problema inițială.

Complexitatea în timp este $\Theta(n * \lg(n))$.

Merge Sort are o performanță mai predictibilă și are întotdeauna o complexitate de timp de O(n log n), făcându-l adecvat pentru liste mari. Ambele algoritmi sunt exemple ale tehnicii "Divide et Impera" și au aplicații importante în sortarea și procesarea datelor.

Codul programului vezi în ANEXA B

Complexitatea de timp și spațiu

Complexitățile de timp și spațiu pentru toți cei trei algoritmi se poate observa în Tabel 1

Tabel 1: Complexitatea de timp

Version	Comlexitatea
Quick Sort	$\Theta(n * \lg(n))\ sau\ \Theta(n^2)$
Merge Sort	$\Theta(n * \lg(n))$.

Time Complexity Quick Sort:

- Best Case: $\Theta(n * \lg(n))$ (când pivotul împarte array-ul în două jumătăți aproximativ egale;
- **Average Case**: $\Theta(n * \lg(n))$;
- **Worst Case**: $\Theta(n^2)$ (apare atunci când cel mai mic sau cel mai mare element este ales întotdeauna ca pivot, conducând la partiții dezechilibrate.)

Time Complexity Merge Sort:

- **Best Case**: $\Theta(n * \lg(n))$;
- **Average Case**: $\Theta(n * \lg(n))$;
- **Worst Case**: $\Theta(n * \lg(n))$.

Space Complexity Quick Sort:

- Space Complexity: $O(logn)$ (datorită stivei de recursie în cazurile cele mai bune și medii; poate fi $O(n)$ în cel mai rău caz dacă este implementat prost.)

Space Complexity Merge Sort:

- Space Complexity: $O(n)$ (necesită spațiu suplimentar pentru array-urile temporare folosite în procesul de interclasare.)

Execuția programului

În imaginea de mai jos putem observa timpul de execuție pentru sortarea a 10000 de elemente în primul rând este timpul de sortare pe merge sort iar în al doilea rând prin quick sort.

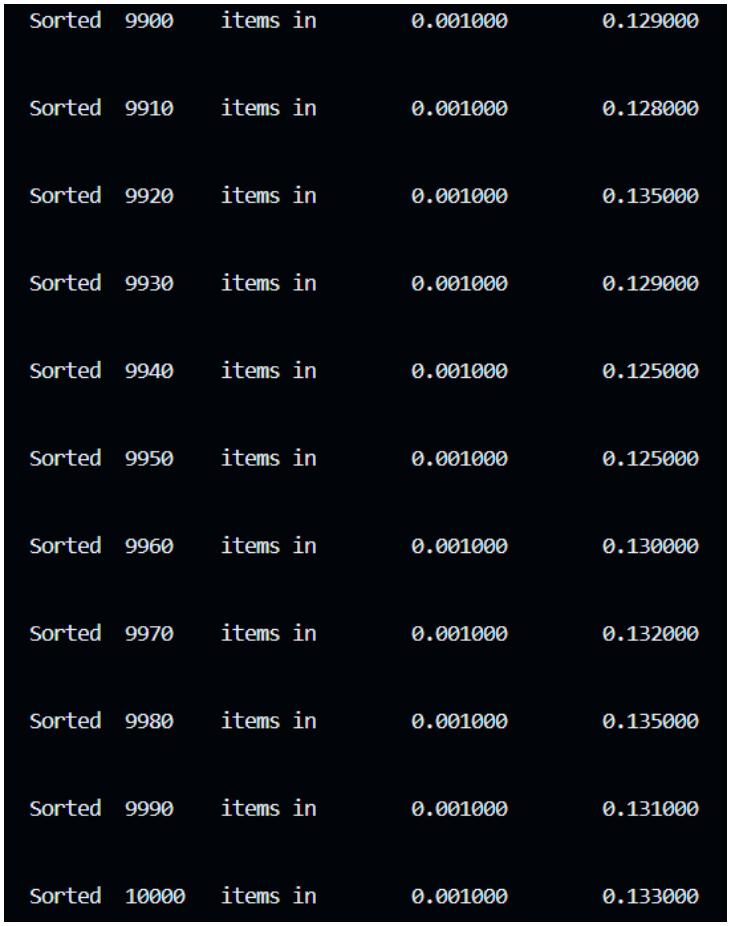


Figura 1: Execuția programului

Analiza empirică

Analiza empirică poate fi observată în tabelul de mai jos Tabel 2.

Tabel 2: Analiza empirică

Algoritmul	Numărul de elemente				
	1000	2500	5000	8000	10000
Quick Sort	0.000105	0.000245	0.000545	0.000839	0.001215
Merge Sort	0.000093	0.000244	0.000505	0.000914	0.001042

Tabelul prezintă analiza empirică a timpului de execuție pentru doi algoritmi diferiți de sortare: algoritmul Quick Sort și algoritmul Merge Sort. Timpul de execuție este măsurat în secunde pentru valori de intrare ale numerelor 1000, 2500, 5000, 8000 și 10000.

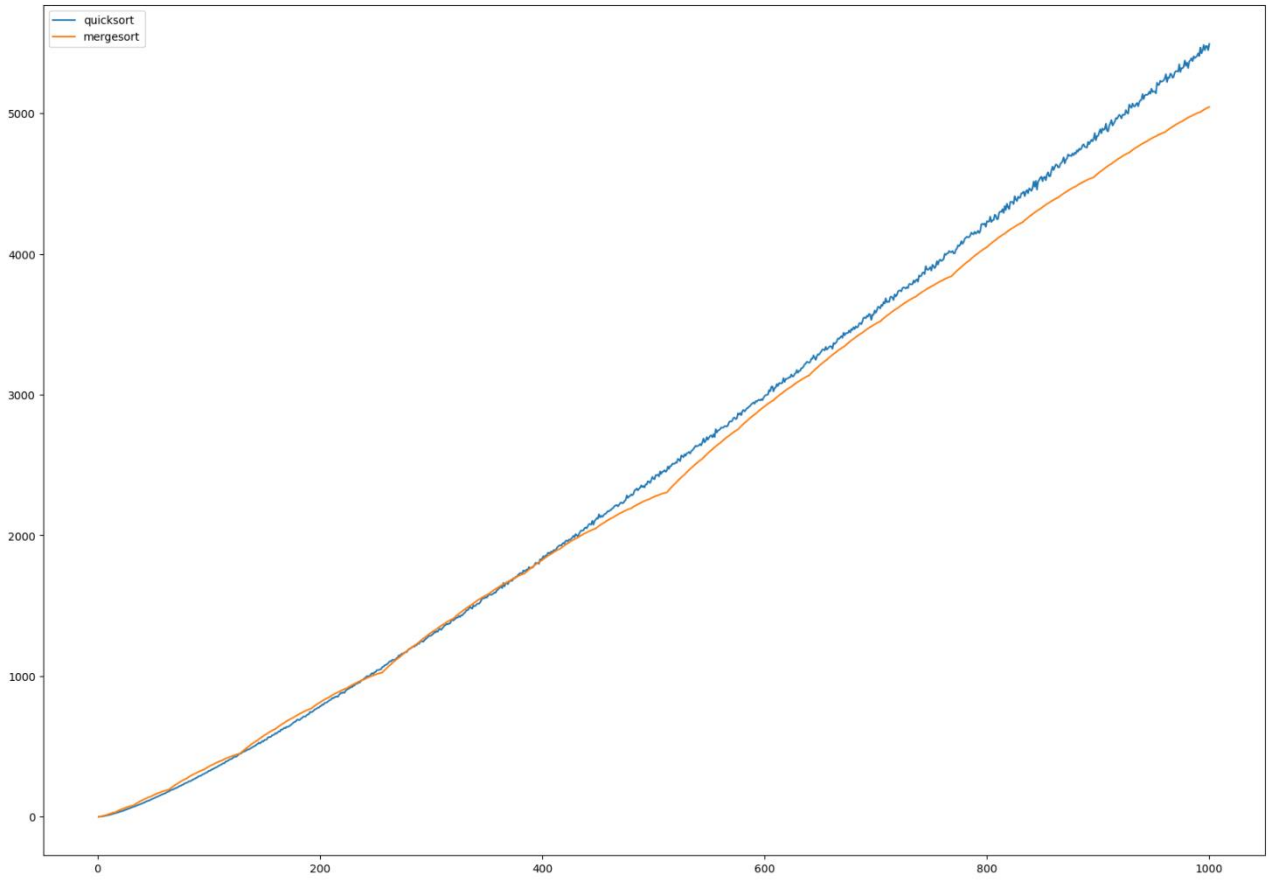
Quick Sort se comportă surprinzător de rău în acest caz, cu timpuri de execuție semnificativ mai mari decât celelalte două algoritme. Acest lucru se datorează faptului că Quick Sort este sensibil la alegerea pivotului și poate avea performanțe slabe pe liste care sunt deja sortate.

Merge Sort oferă performanțe solide în acest caz, cu timpuri de execuție mai bune decât Quick Sort. Merge Sort este cunoscut pentru performanța sa constantă și, în general, funcționează bine pe liste de orice tip.

Quick Sort are o complexitate spațială mai bună decât Merge Sort deoarece Merge Sort necesită spațiu de memorie suplimentar pentru a crea array-uri temporare în timpul procesului de interclasare. Când se interclasează două sub-array-uri, este nevoie de un array temporar pentru a ține elementele interclasate înainte de a fi plasate înapoi în array-ul original.

Grafic de reprezentare a vitezei de execuție

Atât Merge Sort, cât și Quick Sort sunt algoritmi de sortare eficienți, având o complexitate medie a timpului de O(n log n). În general, Quick Sort este mai rapid în practică deoarece are factori constanți mai mici și o performanță mai bună în ceea ce privește cache-ul. Cu toate acestea, Merge Sort este un algoritm stabil, ceea ce înseamnă că păstrează ordinea relativă a elementelor cu chei egale, în timp ce Quick Sort nu este. În plus, Merge Sort are avantajul de a fi ușor de paralelizat. În cele din urmă, alegerea între cele două va depinde de cerințele specifice ale sarcinii în cauză.



Figură 2:Timpul de executare bazat la numărul de iterații

Concluzie

Să discutăm în ce cazuri ar trebui să alegem QuickSort în loc de MergeSort.

Deși atât QuickSort, cât și MergeSort au o complexitate medie a timpului de O(n log n), QuickSort este algoritmul preferat, deoarece are o complexitate a spațiului de O(log(n)). MergeSort, pe de altă parte, necesită O(n) spațiu suplimentar, ceea ce îl face destul de costisitor pentru array-uri.
QuickSort necesită accesarea diferitelor indici pentru operațiile sale, dar acest acces nu este posibil în mod direct în listele legate, deoarece nu există blocuri continue; prin urmare, pentru a accesa un element, trebuie să iterăm prin fiecare nod de la începutul listei legate. De asemenea, MergeSort este implementat fără spațiu suplimentar pentru listele legate.

În acest caz, suprasarcina crește pentru QuickSort și, în general, MergeSort este preferat.

În concluzie, alegerea algoritmului de sortare depinde de dimensiunea și caracteristicile setului de date, precum și de importanța stabilității sortării. Pentru seturi de date mari și performanță maximă, Merge Sort este o opțiune excelentă. Quick Sort oferă un echilibru între eficiență și complexitate, dar poate varia în funcție de distribuția datelor.

Dacă ai nevoie de un algoritm de sortare stabil și poți suporta spațiul suplimentar necesar, Merge Sort poate fi alegerea mai bună. Dacă cauți viteză și lucrezi cu seturi mari de date unde performanța în cazul mediu este critică, Quick Sort este, în general, preferat, cu condiția să fie implementat astfel încât să se evite scenariile din cel mai rău caz (de exemplu, prin alegerea unui pivot bun).

Bibliografie

1. „Curs 12 – Tehnici de programare UBB” https://www.cs.ubbcluj.ro/~istvanc/fp/curs/Curs12%20%20Backtracking%20Divide%20and%20Conquer.pdf
2. Week 5-b: Merge Sort and Quick Sort https://sbme-tutorials.github.io/2020/data-structure-FALL/notes/week05b.html
3. quick-sort-algorithm https://github.com/bradtraversy/traversy-js-challenges/blob/main/09-sorting-algorithms/10-quick-sort-algorithm/readme.md
4. Which algorithm is better between Quick sort and Merge sort with respect to time and space complexity? https://www.quora.com/Which-algorithm-is-better-between-Quick-sort-and-Merge-sort-with-respect-to-time-and-space-complexity
5. „Comparison of Merge Sort and Quick Sort | Intro to Algorithms Class Notes | Fiveable” https://library.fiveable.me/introduction-algorithms/unit-4/comparison-merge-sort-quick-sort/study-guide/hsJZOCKsBrbgPViW
6. „Algorithms and Data Structures.”GitHub https://github.com/davide94/Algorithms-and-Data-Structures

ANEXA A

Codul pentru Merge Sort

```
void merge_sort(int[], int, int);
void merge(int[], int, int, int);

void merge_sort(int A[], int p, int r) {

    if(p<r) {
        int q = (p+r)/2;
        merge_sort(A, p, q);
        merge_sort(A, q+1, r);
        merge(A, p, q, r);
    }
}

void merge(int A[], int p, int q, int r) {

    int n1 = q - p + 1;
    int n2 = r - q;
    int L[n1+1];
    int R[n2+1];

    for (int i=0; i<n1; i++) {
        L[i] = A[p+i];
    }

    for (int i=0; i<n2; i++) {
        R[i] = A[q+i+1];
    }

    L[n1] = R[n2] = MAX + 1;
    int i = 0;
    int j = 0;

    for (int k=p; k<=r; k++) {
        if(L[i] <= R[j]) {
            A[k] = L[i];
```

```
            i++;
        } else {
            A[k] = R[j];
            j++;
        }
    }
}

void merge_sort_worst_case(int A[], int len) {
    for (int i = 0; i < len; i++) {
        A[i] = len - i;
    }
}
```

Anexa B

Codul pentru Quick sort

```
void quick_sort(int[], int, int);
int partition(int[], int, int);

void quick_sort(int A[], int p, int r) {
    if (p <= r) {
        int q = partition(A, p, r);
        quick_sort(A, p, q-1);
        quick_sort(A, q+1, r);
    }
}

int partition(int A[], int p, int r) {
    int x = A[r];
    int i = p - 1;
    for (int j=p; j<=r-1; j++) {
        if (A[j] <= x) {
            i++;
            int swap = A[i];
            A[i] = A[j];
            A[j] = swap;
        }
```

```
    }
    int swap = A[i+1];
    A[i+1] = A[r];
    A[r] = swap;
    return i + 1;
}

void quick_sort_worst_case(int A[], int len) {
    for (int i = 0; i < len; i++) {
        A[i] = i;
    }
}
```

Anexa C

Metoda main

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX 10000

#include "merge_sort.c"
#include "quick_sort.c"

void print_array(int*, int);
void avg();
void worst_case();

int main() {

    avg();
    worst_case();

    return 0;
}

void avg() {
```

```
    FILE *h;
    h = fopen("avg.dat","w");

    for (int len = 10; len <= MAX; len+=10) {

        int *B = malloc(len*sizeof(int));
        int *D = malloc(len*sizeof(int));

        srand(time(NULL));

        for (int i=0; i<len; i++) {
            int value = rand();
            B[i] = value;
            D[i] = value;
        }

        clock_t start, end;
        double  cpu_time_merge_sort, cpu_time_quick_sort;

        start = clock();
        merge_sort(B, 0, len-1);
        end = clock();
        cpu_time_merge_sort = ((double) (end - start)) / CLOCKS_PER_SEC;

        printf("\nSorted    %i  items in    %f  %f \n\n", len, cpu_time_merge_sort,
cpu_time_quick_sort);
        fprintf(h,"%i,%f,%f,%f,%f\n", len, cpu_time_merge_sort, cpu_time_quick_sort);

    }

    fclose(h);
}

void worst_case() {

    FILE *h;
    h = fopen("worst_case.dat","w");
```

```
    for (int len = 10; len <= MAX; len+=10) {

        int *B = malloc(len*sizeof(int));
        int *D = malloc(len*sizeof(int));

        merge_sort_worst_case(B, len);
        quick_sort_worst_case(D, len);

        clock_t start, end;
        double cpu_time_merge_sort, cpu_time_quick_sort;

        start = clock();
        merge_sort(B, 0, len-1);
        end = clock();
        cpu_time_merge_sort = ((double) (end - start)) / CLOCKS_PER_SEC;

        start = clock();
        quick_sort(D, 0, len-1);
        end = clock();
        cpu_time_quick_sort = ((double) (end - start)) / CLOCKS_PER_SEC;


        printf("\nSorted    %i  items in    %f  %f  \n\n", len, cpu_time_merge_sort,
cpu_time_quick_sort);
        fprintf(h,"%i,%f,%f,%f,%f\n", len, cpu_time_merge_sort, cpu_time_quick_sort);

    }

    fclose(h);
}

void print_array(int A[], int len) {

    for (int i = 0; i < len-1; i++) {
        printf("%i, ", A[i]);
    }
    printf("%i\n", A[len-1]);

}
```