O COMPARAȚIE TEORETICĂ ȘI EXPERIMENTALĂ A UNOR METODE DE SORTARE

Rancov Larisa
Facultatea de Matematică și Informatică
Universitatea de Vest Timișoara, România
larisa.rancov03@e-uvt.ro

Rezumat

Lucrarea de față expune o analiză a performanței a șapte metode de sortare: selecție, inserție, interschimbare, heap, quicksort, merge și radix.

Scopul lucrării este de a compara aceste metode în ceea ce privește numărul de comparații, timpul de execuție, timpul de rulare, memoria necesară și numărul de mutări pentru a sorta un set de date de dimensiuni diferite.

Autoarea a realizat și experimente pentru a confirma teoriile, iar concluziile indică faptul că Quicksort este cea mai eficientă metodă din cele șapte analizate.

1 Context și motivație

Una dintre operațiile fundamentale în majoritatea aplicațiilor de calcul, de la bazele de date și până la algoritmii de inteligență artificială, este sortarea. Scopul acestei lucrări este de a compara teoretic si experimental mai multe metode de sortare.

Ținând cont de faptul că sortarea este o problema significantă în domeniul informaticii, deoarece exista o serie numeroasă de aplicații care necesită ordonarea unui set de date, dar și pentru că sortarea poate fi utilizată ca o componentă a algoritmilor de calcul, cercetarea continuă a metodelor de sortare este importanta pentru a îmbunătăți performanța și eficiența acestora.

În ceea ce privește algoritmii de sortare, fiecare metodă disponibilă are propriile avantaje si dezavantaje, iar alegerea corectă a metodei de sortare se bazează pe mai mulți factori, precum modul de implementare, dimensiunea setului de date sau spațiul de memorie alocat. Această lucrare se va concentra pe compararea următoarelor metode de sortare: sortarea prin inserție, sortarea prin selecție, sortarea prin interschimbare, sortarea rapidă, sortarea Heap, sortarea Radix și sortarea Merge.

Pe parcursul acestei lucrări, se vor discuta proprietățile si caracteristicile fiecărei metode de sortare. De asemenea, se va explica analiza teoretică a performanței fiecărei metode, fundamentată prin analiza experimentală.

Pentru a evidenția funcționalitatea algoritmilor de sortare, se vor utiliza trei seturi de date diferite, de 100, 1000 și 10000 de elemente, împreună cu alte trei liste a câte 50000 de elemente, una sortată complet, a doua sortată parțial, cea de-a treia fiind nesortată. Se va efectua sortarea fiecărui set de date și se va analiza performanța fiecărui algoritm în funcție de timpul de execuție și cel de rulare.

Contribuțiile autoarei în realizarea acestei lucrări se referă la comparația teoretică a celor șapte metode de sortare diferite. În același timp, autoarea va efectua experimente pentru a compara performanța lor și în practică.

În continuare, se vor ilustra fundamentarea teoretică a metodelor de sortare, implementarea și evaluarea performanței metodelor de sortare în diferite medii, analiza comparativă a metodelor de sortare pe baza experimentelor, comparația cu literatura, dar și concluzii și sugestii pentru îmbunătățirea performanței metodelor de sortare. Lecturarea lucrării în ordinea prezentată va oferi cel mai bun context și înțelegere a subiectului tratat.

2 Fundamentarea teoretică a metodelor de sortare

Sortarea este definită ca fiind procesul de a aranja un set de elemente într-o anumită ordine. Contextul în care apare această problemă este cel al algoritmilor si structurilor de date, unde sortarea este o operațiune frecvent utilizată în diverse aplicații. Mai exact, problema sortării poate fi definită formal prin următoarele elemente:

- Setul de elemente ce trebuie sortat: $A = a1, a2, ..., an$, unde fiecare ai este un element dintr-un set ordonat total.
- Ordinea de sortare: Aceasta poate fi definită printr-un criteriu, cum ar fi sortarea crescătoare, sau printr-o funcție de comparație definită pentru elementele din set.
- Soluția problemei: Soluția problemei de sortare este o permutare a elementelor din setul A, astfel încât elementele sunt aranjate în ordinea specificată. O soluție poate fi reprezentată prin vectorul sortat $B = b1, b2, ..., bn$, unde fiecare bi este un element din setul A.

Pentru a formaliza soluția problemei, se pot analiza complexitatea fiecărui algoritm și eficiența sa. Astfel, se vor descrie formal numărul de comparații și mutări necesare pentru a sorta lista de elemente. De asemenea, se poate evalua performanța algoritmului prin timpul de execuție și memoria necesară pentru a sorta lista.

În continuare, se vor folosi notații și caracteristici ale algoritmilor de sortare ilustrate în lucrările [1],[2],[4],[3]:

- Sortarea prin inserție

  Numărul de comparații: cel puțin $n - 1$ și de cel mult $\frac{n \times (n-1)}{2}$, unde n este numărul de elemente din listă.

  Numărul de mutări: cel puțin $n - 1$ și de cel mult $\frac{n \times (n-1)}{2}$, unde n este numărul de elemente din listă.

  Timpul de execuție: depinde de numărul de elemente din listă și de ordinea inițială a elementelor.Cel mai rău caz este atunci când lista este sortată în ordine inversă, iar timpul de executie pentru acest caz este $O(n^2)$, unde n este dimensiunea listei. Cel mai bun caz este atunci când lista este deja sortată, iar timpul de executie pentru acest caz este O(n).

  Memoria necesară: depinde de numărul de elemente din listă și de dimensiunea elementelor.

- Sortarea prin selecție

  Numărul de comparații: $\frac{n^2-n}{2}$ (cel mai rău caz, pentru un array de dimensiune n).

  Numărul de mutări: $O(n)$ (cel mai bun și cel mai rău caz) .

  Timpul de execuție: $O(n^2)$ (cel mai rău caz).

  Memoria necesară: $O(1)$ (sortarea se face în loc).

- Sortarea prin interschimbare

  Numărul de comparații:

  - Cel mai bun caz: $n-1$ comparații, când lista este deja sortată;
  - Cel mai rău caz: $n*(n-1)/2$ comparații, când lista este sortată invers;
  - Cazul mediu: $n*(n-1)/4$ comparații.

  Numărul de mutări:

  - Cel mai bun caz: 0 mutări, când lista este deja sortată;
  - Cel mai rău caz: $n*(n-1)/2$ mutări, când lista este sortată invers;
  - Cazul mediu: $n*(n-1)/4$ mutări.

  Timpul de execuție:

  - Cel mai bun caz: $O(n)$, când lista este deja sortată;
  - Cel mai rău caz: $O(n^2)$, când lista este sortată invers;
  - Cazul mediu: $O(n^2)$.

  Memoria necesară: O(1), deoarece interschimbările se fac între elemente adiacente și nu este necesară alocarea suplimentară de memorie

- Sortarea Heap

  Numărul de comparații: în cel mai rău caz, algoritmul Heap necesită $O(n*\log n)$ comparații.

  Numărul de mutări: în cel mai rău caz, algoritmul Heap necesită $O(n*\log n)$ mutări pentru a sorta un vector de dimensiune n.

  Timpul de execuție: în cel mai rău caz, algoritmul Heap necesită $O(n*\log n)$ timp pentru a sorta un vector de dimensiune n.

  Memoria necesară: algoritmul Heap necesită $O(1)$ memorie suplimentară pentru a sorta vectorul de intrare, deoarece sortarea se face direct în vectorul dat ca intrare.

- Sortarea Radix

  Numărul de comparații: $O(n * k)$, unde n este numărul de elemente din lista de sortat și k este numărul maxim de cifre din cheile numerice.

  Numărul de mutări: depinde de numărul de cifre ale elementelor din listă. Dacă numărul maxim de cifre este k, atunci numărul de mutări este $O(n * k)$, unde n este numărul de elemente din listă.

  Timpul de execuție: $O(n * k)$. Acesta este unul dintre cele mai rapide algoritmi de sortare pentru chei numerice de dimensiuni mici și moderate.

  Memoria necesară: $O(n + k)$

- Sortarea Merge

  Numărul de comparații: $2 * n * \log n$.

  Numărul de mutări: $n * \log n$.

  Timpul de execuție: $O(n * \log n)$, unde n este dimensiunea listei.

  Memoria necesară: $O(n)$.

- Sortarea rapidă

  Numărul de comparații: în medie, are un număr de comparații de aproximativ $1.39 * n * \log_2 n$, unde n este numărul de elemente din lista de sortat.

  Numărul de mutări: în medie, necesită $O(n * \log_2 n)$ mutări.

  Timpul de execuție: în medie, are o complexitate de timp de $O(n * \log_2 n)$ în cel mai rău caz și O(n) în cel mai bun caz.

  Memoria necesară: consumă $O(\log_2 n)$ memorie pentru stiva de apeluri recursive, în cel mai rău caz.

Teoretic, eficiența algoritmilor se deduce din ordinul de complexitate al acestora. Prin reprezentarea fiecărui ordin de complexitate în graficul de mai jos (vezi Figura 1), se ajunge la ideea conform căreia cel mai eficient algoritm de sortare este cel care are timpul de execuție $O(n * \log n)$.

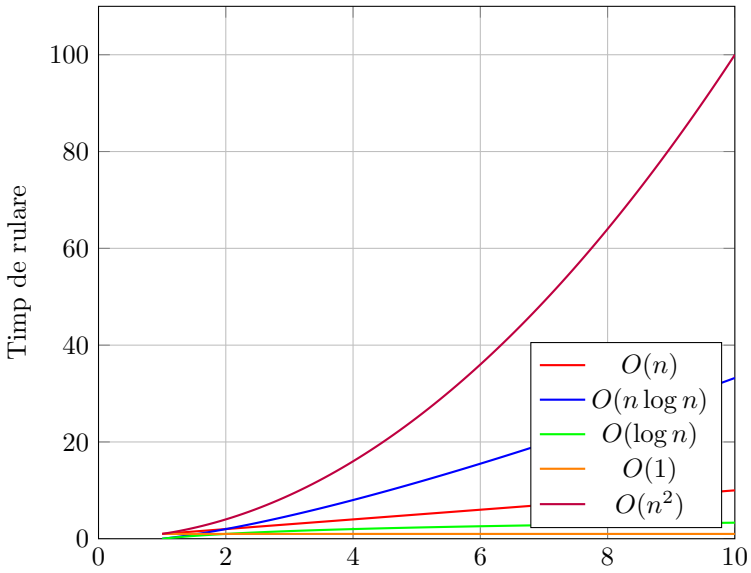

Figura 1. Reprezentarea ordinelor de complexitate

3 Implementarea și funcționalitatea algoritmilor de sortare și a soluției

Limbajul de programare folosit pentru implementarea celor șapte algoritmi de sortare a fost C, iar Codeblocks a reprezentat mediul de dezvoltare utilizat.

În cele ce urmează, se va ilustra implementarea fiecărui algoritm și modul în care poate fi folosită aceasta.

- Sortarea prin inserție

```
void insertion_sort(int arr[], int n) {
    int i, key, j;
    for (i = 1; i < n; i++) {
        key = arr[i];
        j = i - 1;
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

Această implementare a fost preluată și adaptată din [2]. Descrierea modului de funcționare a implementării este următoarea:

a. Se va defini un vector de elemente care trebuie sortate.

b. În cazul sortării în ordine crescătoare, fiecare element este inserat în sub-vectorul sortat înaintea elementelor mai mari. În cazul sortării în ordine descrescătoare, fiecare element este inserat în sub-vectorul sortat înaintea elementelor mai mici.

c. Se parcurge vectorul și se inserează fiecare element în sub-vectorul sortat folosind o buclă while.

d. La sfârșitul buclei, vectorul va fi sortat în ordinea dorită.

- Sortarea prin selecție

```
void selection_sort(int arr[], int n) {
    int i, j, min_idx;
    for (i = 0; i < n - 1; i++) {
        min_idx = i;
        for (j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }
    }
}
```

Această implementare a fost preluată și adaptată din [2]. Descrierea modului de funcționare a implementării este următoarea:

a.Se parcurge lista nesortată și se caută cel mai mic element.

b.Cel mai mic element găsit se schimbă cu primul element din lista nesortată.

c.Se repetă acești pași pentru lista rămasă nesortată, ignorând primul element sortat.

Procesul se repetă până când întreaga listă este sortată. La fiecare iterație, se găsește cel mai mic element și se plasează la locul său corect în listă.

- Sortarea prin interschimbare

```
void bubble_sort(int arr[], int n) {
    int i, j;
    for (i = 0; i < n - 1; i++) {
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1])
                swap(&arr[j], &arr[j + 1]);
        }
    }
}
```

Această implementare a fost preluată și adaptată din [2]. Descrierea modului de funcționare a implementării este următoarea:

a.Se parcurge lista de elemente de la primul până la penultimul element.

b.Pentru fiecare element, se parcurge lista de elemente rămase de la primul până la cel mai mare element de pe poziția curentă - 1.

c.Dacă elementul curent este mai mare decât următorul element, se face interschimbarea între cele două elemente.

d.Se repetă pașii 2-3 până când lista este sortată.

e.La finalul procesului, lista este sortată în ordine crescătoare.

- Sortarea Heap

```
void heapify(int arr[], int n, int i) {
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;
    if (l < n && arr[l] > arr[largest])
        largest = l;
    if (r < n && arr[r] > arr[largest])
        largest = r;
    if (largest != i) {
        swap(&arr[i], &arr[largest]);
        heapify(arr, n, largest);
    }
}

void heap_sort(int arr[], int n) {
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);
    for (int i = n - 1; i >= 0; i--) {
        swap(&arr[0], &arr[i]);
        heapify(arr, i, 0);
    }
}
```

Această implementare a fost preluată și adaptată din [2]. Descrierea modului de funcționare a implementării este următoarea:

a. Se construiește un arbore Heap (sau subarbore), începând de la ultimul nod, folosind funcția 'heapify'.

b. Se extrage elementele din arborele Heap unul câte unul prin mutarea rădăcinii la sfârșit și aplicarea funcției 'heapify' pe arborele rămas, cu rădăcina la 0 și dimensiunea n = i.

c. Se repetă pasul 2 până când toate elementele sunt sortate în ordine crescătoare.

- Sortarea Radix

```
void radix_sort(int arr[], int n) {
    int i, bucket[10] = {0}, max_value = 0, exp = 1;
    int *aux = (int*) malloc(n * sizeof(int));
    for (i = 0; i < n; i++) {
        if (arr[i] > max_value) {
            max_value = arr[i];
        }
    }
    while (max_value / exp > 0) {
        for (i = 0; i < 10; i++) {
            bucket[i] = 0;
        }
        for (i = 0; i < n; i++) {
            bucket[(arr[i] / exp) % 10]++;
        }
        for (i = 1; i < 10; i++) {
            bucket[i] += bucket[i - 1];
        }
        for (i = n - 1; i >= 0; i--) {
            aux[--bucket[(arr[i] / exp) % 10]] = arr[i];
        }
        for (i = 0; i < n; i++) {
            arr[i] = aux[i];
        }
        exp *= 10;
    }

    free(aux);
}
```

Această implementare a fost preluată și adaptată din [3]. Descrierea modului de funcționare a implementării este următoarea:

a. Se găsește numărul maxim de cifre din vectorul de sortat și se inițializează o matrice de contorizare pentru fiecare cifră (0-9).

b. Se parcurge vectorul de la cifra unităților până la cea mai semnificativă cifră.

c. Se rearanjează vectorul în funcție de matricea de contorizare și cifra curentă, astfel încât să fie ordonat crescător sau descrescător după cifra curentă.

- Sortarea Merge

```
void merge_sort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        merge_sort(arr, l, m);
        merge_sort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
```

Această implementare a fost dezvoltată de către John von Neumann în anii 1940 și a fost descrisă în lucrările sale publicate în 1945. Descrierea modului de funcționare a implementării este următoarea:

Algoritmul de sortare Merge funcționează în următorii pași:

a. Se împarte vectorul de sortat în jumătate până când fiecare sub-vector conține un singur element.

b. Se combină sub-vectorii în perechi și se sortează fiecare pereche. Aceasta se poate face prin compararea primelor elemente din fiecare pereche și inserarea lor în ordine crescătoare într-un nou vector.

c. Dacă numărul de sub-vectori este impar, se alege ultimul sub-vector și se copiază în noul vector.

d. Se repetă pașii 2 și 3 până când nu mai există decât un singur vector, care este vectorul sortat final.

- Sortarea rapidă

```
void quick_sort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        quick_sort(arr, low, pi - 1);
        quick_sort(arr, pi + 1, high);
    }
}
```

Această implementare a fost a fost dezvoltată de către Tony Hoare in 1959 și a fost publicată in 1961 în Communications of the ACM. Descrierea modului de funcționare a implementării este următoarea:

a. Se alege un element pivot din vector.

b. Se împarte vectorul în două sub-vectori, astfel încât elementele din stânga pivotului sunt mai mici decât pivotul, iar cele din dreapta sunt mai mari sau egale cu pivotul.

c. Recursiv, se aplică algoritmul quicksort pe cei doi sub-vectori obținuți în urma împărțirii.

d. Se combină sub-vectorii sortați pentru a obține vectorul sortat final.

Pentru a utiliza implementarea acestor algoritmi de sortare în C, se vor urmări pașii de mai jos:

Se descarcă fișierul cu codul sursă al implementării.

Se deschide editorul de cod și se creează un nou fișier.

Se copiază codul sursă al implementării în fișierul nou creat.

Se salvează fișierul cu o denumire relevantă.

Se compilează codul sursă folosind un compilator C, cum ar fi GCC.

Se execută fișierul generat de compilator pentru a rula programul.

Se introduc valorile care trebuie sortate și se verifică rezultatul sortării.

Modalitatea prin care autoarea a generat cele 4 liste pe care le-a comparat folosind algoritmii descriși mai sus a fost următoarea:

```
import random
l = [random.randint(0,100000) for i in range(1000)]
print(l)
```

Astfel, pentru fiecare listă s-a introdus numărul dorit de elemente, iar funcția random a generat elementele într-o ordine aleatorie, fiind pregătite pentru sortare.

4 Analiza comparativă a metodelor de sortare pe baza experimentelor

În ceea ce privește analiza experimentală a algoritmilor de sortare menționați anterior, autoarea a decis să compare metodele calculând timpii de rulare utilizând funcția clock() din librăria time.h .

Prima parte a experimentului s-a desfășurat în felul următor:

S-a sortat atât parțial, cât și complet lista de 50000 de elemente.

S-a calculat timpul de rulare pentru fiecare algoritm în parte, în cazul listei nesortate (cazul nefavorabil), al listei sortate parțial(cazul mediu) și al listei sortate complet(cazul favorabil).

Table 1: Compararea timpilor de executie ai algoritmilor de sortare

Algoritmul	Caz favorabil	Caz mediu	Caz nefavorabil
Inserția	0.07	1.906	2.455
Selecția	2.779	2.803	2.861
Heap	0.003	0.007	0.016
BubbleSort	2.644	6.34	7.632
Merge	0.008	0.016	0.016
Quick	Un număr ciudat	0.006	0.015
Radix	0.014	0.005	0.019

Conform rezultatelor obținute, se poate constata faptul că algoritmul de sortare Quicksort are o performanță mai bună decât ceilalți algoritmi, în timp ce Bubblesort este cel mai puțin eficient. Din analiza datelor, se poate concluziona faptul că QuickSort este cel mai eficient algoritm de sortare, urmat îndeaproape de Radix și Heap Sort. În ceea ce privește testarea unor liste deja sortate cu algoritmul Quicksort, s-a observat un comportament neașteptat, care sugerează că acest algoritm nu răspunde prea bine la cazurile degenerate. În timpul testării algoritmilor Bubblesort, Selection Sort și Insertion sort, au fost întâmpinate dificultăți la compilare, deoarece listele nu au început să se sorteze instant, ci a durat puțin până au apărut numerele în fereastra de compilare. În comparație, pentru ceilalți algoritmi, s-a observat o compilare mai facilă, ceea ce sugerează că aceștia sunt mai ușor de utilizat. Rezultatele obținute au fost concluzive, deoarece setul de date ales a fost unul complex.

A doua parte a experimentului a avut loc în următorul mod:

S-a calculat timpul de rulare al celor patru liste de 100, 1000, 10000 și 50000 de elemente.

S-au comparat rezultatele.

Table 2: Compararea algoritmilor de sortare

	Număr de elemente			
	100	1000	10000	50000
Insertion Sort	0,002	0,006	0,008	2,455
Selection Sort	0,002	0,006	0,007	2,861
Merge Sort	0,0001	0,0002	0,004	0,016
QuickSort	0,0001	0,0001	0,003	0,015
Heap Sort	0,0001	0,0003	0,001	0,016
Bubblesort	0,001	0,011	0,037	7,632
Radix Sort	0,0001	0,0001	0,006	0,019

Din tabelul prezentat mai sus, se poate observa că toți algoritmii de sortare funcționează eficient în cazul listelor cu o cantitate redusă de date de intrare. Cu toate acestea, odată cu creșterea numărului de elemente din listă, timpul de rulare se modifică semnificativ. Conform datelor din tabel, algoritmul Bubblesort înregistrează un timp de rulare de 0,011, în timp ce sortarea prin inserție și selecție au un timp de rulare de 0,006.

Prin compararea rezultatelor obținute prin testarea listelor de 100 și 1000 de numere, se poate deduce faptul că nu există diferențe semnificative între timpii de rulare ai algoritmilor. În același timp, concluzia conform căreia Bubblesort, Selection Sort și Insertion Sort sunt cele mai puțin eficiente metode de sortare se poate confirma și prin datele obținute în practică.

Odată cu testarea listei de 10000 de numere, rezultatele s-au modificat semnificativ, atât pentru algoritmii menționați anterior, cât și pentru Merge Sort și Heap Sort. Prin urmare, aceștia nu mai pot fi considerați printre cei mai eficienți algoritmi de sortare. În ciuda faptului că modificările nu sunt foarte semnificative, acestea pot avea un impact important în ceea ce privește alegerea unui algoritm de sortare adecvat.

La testarea listei de 50000 de numere, diferențele de eficiență dintre algoritmi devin extrem de clare. Din cei doi algoritmi rămași, Quicksort și Radixsort, Quicksort este considerat a fi cel mai util, datorită timpului său de rulare cu 0,003 mai mic față de Radixsort.

5 Comparație cu literatura

Experimentul din această lucrare a avut ca scop analiza și compararea algoritmilor de sortare în funcție de timpii de execuție și de cei de rulare.

În literatura de specialitate, volumul lui Donald Knuth, intitulat 'The Art of Computer Programming', este considerat unul dintre cele mai importante în domeniu. Acesta a realizat o analiză detaliată a complexității teoretice a algoritmilor de sortare și a constatat că performanța algoritmilor de sortare poate varia semnificativ în funcție de dimensiunea datelor de intrare, de distribuția datelor și de implementarea specifică a algoritmului.

În comparație cu lucrarea de față, lucrarea lui Donald Knuth prezintă mai multe modalități de măsurare a eficienței, însă ambele ilustrează o concluzie comună, aceea că sortarea rapidă este considerată una dintre cele mai rapide și eficiente metode de sortare.

6 Concluzii și sugestii pentru îmbunătățirea performanței metodelor de sortare

Pe parcursul lucrării s-a realizat analiza teoretică a șapte algoritmi de sortare, anume sortarea prin inserție, selecție, interschimbare, sortarea rapidă, heap, radix și merge. S-a constatat așadar faptul că ordinul de complexitate este cel care exprimă teoretic eficiența unui algoritm, $O(n * \log n)$ fiind ordinul de complexitate care reprezintă cei mai eficienți algoritmi precum QuickSort, MergeSort sau HeapSort.

De asemenea, s-a înfăptuit și o analiză practică, a timpilor de rulare. În urma acestei analize, QuickSort a fost nominalizat drept cel mai eficient algoritm de sortare, având cel mai mic timp de rulare. Totodată, acest algoritm nu funcționează conform așteptărilor în cazul unei liste deja complet sortate, ceea ce duce la deznodământul acestei lucrări: pentru a obține performanțe cât mai optime, este necesar să se țină cont de caracteristicile datelor care urmează a fi sortate și să se aleagă metoda de sortare adecvată.

Problema sortării și găsirea unor rezolvări cât mai eficiente a acesteia este un subiect important și continuă să fie cercetat în comunitatea informatică. Analiza atentă a avantajelor și dezavantajelor fiecărei metode, împreună cu optimizarea implementării, poate duce la îmbunătățirea semnificativă a performanței și poate face diferența în aplicațiile practice.

References

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[3] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 1998.

[4] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley Professional, 4th edition, 2011.