O comparație teoretică și experimentală a mai multor algoritmi de sortare

Simina-Elena Fildan*

12 mai 2024

Rezumat

Sortarea rămâne o problemă de actualitate în informatică, fiind o operațiune frecventă în manipularea și gestionarea datelor. Eficiența sortării reprezintă un factor esențial în dezvoltarea și optimizarea algoritmilor.

În acest articol, ne propunem să comparăm diverse metode de sortare, evidențiind avantajele și dezavantajele fiecăreia și oferind o perspectivă clară asupra criteriilor de selecție a unei metode potrivite în diverse contexte de utilizare. Vom investiga aspecte precum complexitatea temporală și spațială, comportamentul avut în funcție de seturi de date de diferite mărimi, oferind cititorului o perspectivă largă asupra avantajelor și dezavantajelor utilizării fiecărui algoritm prezentat.

Ne vom opri asupra a nouă algoritmi de sortare consacrați:

- Sortarea prin inserție
- Sortarea prin selecție
- Sortarea prin interschimbarea elementelor vecine
- Sortarea prin interclasare
- Sortarea rapidă
- Sortarea prin numărare
- Sortarea pe baza cifrelor
- Shell Sorting
- Shaker Sort

Cuvinte cheie: sortare, algorimică

*Univeritatea de Vest din Timișoara, Facultatea de Matematică și Informatică, Departamentul de Informatică, E-mail:simina.fildan04@e-uvt.ro

Cuprins

1 Introducere

1.1 Motivație

Sortarea rămâne una dintre cele mai fundamentale și omniprezente operațiuni în domeniul informaticii. De la gestionarea bazelor de date la analiza datelor în domenii precum inteligența artificială și învățarea automată, eficiența sortării este esențială pentru performanța și funcționalitatea adecvată a sistemelor informatice.

Sortarea, în ciuda aspectului său aparent simplu, este o problemă complexă, care implică o varietate de factori, cum ar fi cantitatea de date, tipul datelor și condițiile de performanță. Selectarea și implementarea unui algoritm de sortare potrivit pentru o anumită aplicație pot fi lucruri dificile, care necesită o înțelegere profundă a caracteristicilor și performanțelor algoritmilor existenți.

Această lucrare se concentrează pe analiza și compararea mai multor algoritmi de sortare, urmărind să ofere o perspectivă comprehensivă asupra diverselor tehnici de sortare existente. Scopul este de a evidenția avantajele și dezavantajele fiecărei metode, astfel încât cititorul să poată lua decizii informate în alegerea algoritmului potrivit pentru nevoile sale specifice.

Prin abordarea atât teoretică, cât și experimentală, lucrarea explorează complexitatea temporală și spațială a fiecărui algoritm de sortare menționat anterior, analizând comportamentul acestora în diferite contexte de utilizare.

1.2 Declarație de originalitate

Declar că eventualele contribuții aduse de alte persoane, precum algoritmii de sortare și proprietățile acestora, sunt explicit recunoscute și citate în mod corespunzător în text și în bibliografia lucrării. Contribuția mea constă în implementarea algoritmilor, în generarea seturilor de date, dar și în analizarea și interpretarea rezultatelor.

1.3 Prezentare structură

Lucrarea este structurată în următoarele secțiuni, fiecare abordând aspecte specifice ale analizei și comparării algoritmilor de sortare. În **Secțiunea 1 - Introducere**, prezentăm contextul și motivația cercetării, oferind o descriere succintă a obiectivelor lucrării. Următoarea secțiune, **Secțiunea 2 - Fundamentare teoretică**, este dedicată unei prezentări a algoritmilor de sortare investigați. Aici vor fi descrise principalele metode de sortare analizate, împreună cu aspectele teoretice și caracteristicile fiecăruia. În **Secțiunea 3 - Modelare și implementare**, detaliem modul în care am implementat algoritmii de sortare. **Secțiunea 4 - Rezultate și analiză** prezintă rezultatele obținute în urma testelor și experimentelor realizate. **Secțiunea 5 - Comparația cu literatura** are ca scop plasarea rezultatelor în contextul comunității și recunoașterea contribuțiilor altor persoane. În **Secțiunea 6 - Concluzii și direcții viitoare**, se trasează concluziile lucrării și se sugerează direcții pentru viitoare cercetări în domeniu.

2 Fundamentare teoretică

În această lucrare, vom compara din punct de vedere teoretic și experimental 9 algoritmi de sortare consacrați: sortarea prin inserție, sortarea prin selecție, sortarea prin interschimbarea elementelor vecine, sortarea prin interclasare, sortarea rapidă, sortarea prin numărare, sortarea pe baza cifrelor, Shell Sorting și Shaker Sort.

La sortarea prin inserție, fiecare element din tablou, începând cu al doilea, este inserat în subtabloul deja ordonat din fața lui (subtablou destinație), astfel încât acesta să rămână sortat. Nu se cunoaște exact cine a inventat acest algoritm, dar este descris de Sedgewick în [Sedgewick(1990)].

În cazul sortării prin selecție, pentru fiecare poziție i, începând cu prima, se caută minimul din elementele aflate între pozițiile i și n, minimul găsit interschimbându-se cu elementul de pe poziția i. La fel ca în cazul sortării prin inserție, autorul algoritmului este necunoscut, dar detalii despre funcționalitate se găsesc în [Sedgewick(1990)].

Ideea de bază a sortării prin interschimbare, [Friend(1956)], constă în compararea elementelor vecine și interschimbarea lor dacă nu sunt în ordinea dorită. Tabloul este parcurs din nou dacă a fost necesară cel puțin o interschimbare în parcurgerea anterioară.

La sortarea prin interclasare, vezi [von Neumann(1982)], se împarte tabloul $a[1...n]$ în două subtablouri: $a[1...n/2]$ și $a[n/2+1...n]$, iar fiecare subtablou este sortat. Apoi, elementele celor doi vectori sunt interclasate pentru a construi tabloul sortat $t[1...n]$.

În cazul sortării rapide, [Hoare(1962)], tabloul $a[1...n]$ este reorganizat și împărțit în două subtablouri. Aceste subtablouri sunt sortate și apoi concatenate.

Sortarea prin numărare, [Seward(1954)], presupune construirea unui tabel de frecvențe pentru elementele din tablou și utilizarea acestuia pentru a construi tabloul ordonat.

Sortarea pe baza cifrelor constă în ordonarea elementelor în funcție de cifra cea mai puțin semnificativă, repetând acest proces până când se ajunge la cifra cea mai semnificativă. Acest algoritm a fost creat de Herman Hollerith, dar nu am găsit lucrarea originală a autorului. Totuși, este descris în [Sedgewick(1990)].

Algoritmul Shell Sorting, [Shell(1959)], este o variantă îmbunătățită a sortării prin inserție, care încearcă să reducă numărul de operații efectuate prin compararea și deplasarea elementelor la distanțe mai mari.

Shaker Sort este o variantă eficientă a interschimbării elementelor vecine, în care la fiecare etapă se plasează pe pozițiile finale câte două elemente (minimul și maximul din subtablou). Originea exactă a algoritmului este necunoscută, fiind explicat în [Sedgewick(1990)].

Următoarele proprietăți ale algoritmilor de sortare sunt preluate din [Sedgewick(1990)] și din [Cormen and Leiserson(2009)].

Toate metodele de sortare descrise sunt corecte, proprietate ce poate fi demonstrată prin găsirea unui invariant și a unei funcții de terminare pentru fiecare ciclu.

Sortarea prin inserție este stabilă (păstrează ordinea elementelor care au aceeași valoare a cheii de sortare), la fel și sortarea prin interschimbarea elementelor vecine (dacă în algoritm se folosește inegalitatea strictă pentru compararea elementelor vecine). Sortarea prin interclasare este stabilă dacă în etapa de interclasare se folosește $\leq$. Counting sort, Radix Sort și Shaker

Sort sunt stabili. Sortarea prin selecție, sortarea rapidă, Shell Sort nu sunt instabile.

Primele 3 metode de sortare amintite (inserție, selecție, interschimbare) sunt simple, ușor de înțeles, însă nu la fel de eficiente ca metodele mai avansate. La sortarea prin inserție, cel mai favorabil caz este cel în care tabloul este deja ordonat, caz în care sortarea are complexitate liniară. În cazul cel mai defavorabil, dacă elementele sunt în ordine descrescătoare, complexitatea e pătratică. Varianta de Bubble Sort descrisă anterior are complexitate liniară în cel mai favorabil caz și complexitate pătratică în cel mai defavorabil caz. Sortarea prin selecție are complexitate pătratică. Sortarea prin interclasare are complexitatea $n \log n$, însă utilizează un tablou adițional de dimensiunea celui inițial. La algoritmul Quicksort, dacă fiecare partiționare este echilibrată (tabloul e împărțit în două subtablouri de dimensiuni apropiate de $\frac{n}{2}$), avem de-a face cu cazul cel mai favorabil, care are complexitate $n \log n$. În cazul cel mai defavorabil, adică atunci când fiecare partiționare e neechilibrată, complexitatea este pătratică. În cazul mediu, ordinul de complexitate este $n \log n$. Spre deosebire de Merge Sort, spațiul adițional are dimensiune constantă.

Counting Sort are complexitate liniară dacă m are dimensiuni mai mici sau apropiate de n, fiind mai eficient decât Quicksort sau Merge Sort. În schimb, dacă m e mult mai mare decât n, complexitatea este $n + m$. Dacă Radix Sort se apelează pentru $m = 9$, complexitatea este liniară. Pentru ordonarea unui volum mare de date, Radix Sort este în general mai eficient decât alți algoritmi, cum ar fi Quick Sort sau Merge Sort. La fel ca la sortarea prin numărare, este nevoie de spațiu adițional de memorie. Varianta folosită mai sus a algoritmului Shell Sort are ordinul de complexitate $n\sqrt{n}$, deci este mai eficientă decât Bubble, Selection sau Insertion Sort, dar mai puțin eficientă decât Quick sau Merge Sort. La Shaker Sort, la fel ca la Insertion și Bubble Sort, în cazul cel mai favorabil (tablou deja ordonat crescător), complexitatea e liniară, iar în cazul defavorabil complexitatea e pătratică.

3 Modelare și implementare

Implementările ce urmează a fi prezentate se bazează pe pseudocodul din [Sedgewick(1990)] și din [Cormen and Leiserson(2009)].

```
1 void sortinsert(int a[], int n)
2 {
3     int i, j, aux;
4     for(i=1;i<n;i++)
5     {
6         aux=a[i];
```

```
        j=i-1;
        while(j>=0 && aux<a[j])
        {
            a[j+1]=a[j];
            j--;
        }
        a[j+1]=aux;
    }
}
```

Listing 1: Sortarea prin inserție

```
void sort_selectie(int a[], int n)
{
    int i, j, k, aux;
    for(i=1;i<=n-1;i++)
    {
        k=i;
        for(j=i+1;j<=n;j++)
            if(a[k]>a[j])
                k=j;
        if(k!=i)
        {
            aux=a[i];
            a[i]=a[k];
            a[k]=aux;
        }
    }
}
```

Listing 2: Sortarea prin selecție

```
void sort_interclasare(int a[], int st, int dr)
{
    int i, m, j, k;
    if(st<dr)
    {
        m=(st+dr)/2;
        sort_interclasare(a,st,m);
        sort_interclasare(a,m+1,dr);
        i=st; j=m+1; k=0;
        while(i<=m && j<=dr)
            if(a[i]<a[j])
                t[++k]=a[i++];
            else
                t[++k]=a[j++];
        while(i<=m)
            t[++k]=a[i++];
        while(j<=dr)
            t[++k]=a[j++];
```

```
        for(i=st,j=1;i<=dr;i++,j++)
            a[i]=t[j];
    }
}
```

Listing 3: Sortarea prin interclasare

```
void sort_interschimbare(int a[], int n)
{
    int i, j, ok, aux;
    do
    {
        ok=1;
        for(i=1;i<=n-1;i++)
            if(a[i]>a[i+1])
            {
                aux=a[i];
                a[i]=a[i+1];
                a[i+1]=aux;
                ok=0;
            }
    }
    while(!ok);
}
```

Listing 4: Sortarea prin interschimbare

```
int pivot(int a[], int st, int dr)
{
    int v, i, j, aux;
    v=a[dr];
    i=st-1;
    j=dr;
    while(i<j)
    {
        do
            i++;
        while(a[i]<v);
        do
            j--;
        while(a[j]>v);
        if(i<j)
        {
            aux=a[i];
            a[i]=a[j];
            a[j]=aux;
        }
    }
    aux=a[i];
    a[i]=a[dr];
```

```
    a[dr]=aux;
    return i;
}
void sort_rapida(int a[], int st, int dr)
{
    int q;
    if(st<dr)
    {
        q=pivot(a,st,dr);
        sort_rapida(a,st,q-1);
        sort_rapida(a,q+1,dr);
    }
}
```

Listing 5: Sortarea rapidă

```
void sort_numarare(int a[], int n, int m)
{
    int i;
    int f =(int) malloc(sizeof(int)*100000010);
    int y =(int) malloc(sizeof(int)*100000010);
    for(i=1;i<=m;i++)
        f[i]=0;
    for(i=1;i<=n;i++)
        f[a[i]]++;
    for(i=2;i<=m;i++)
        f[i]+=f[i-1];
    for(i=n;i>=1;i--)
    {
        y[f[a[i]]]=a[i];
        f[a[i]]--;
    }
    for(i=1;i<=n;i++)
        a[i]=y[i];
}
```

Listing 6: Sortare prin numărare

```
void counting(int a[], int n, int m, int c)
{
    int i, j;
    int f =(int) malloc(sizeof(int)*100000010);
    int y =(int) malloc(sizeof(int)*100000010);
    for(i=0;i<=m;i++)
        f[i]=0;
    for(i=1;i<=n;i++)
    {
        j=((int)(a[i]/pow(10,c)))%10;
        f[j]++;
    }
```

```
    for(i=1;i<=m;i++)
        f[i]+=f[i-1];
    for(i=n;i>=1;i--)
    {
        j=((int)(a[i]/pow(10,c)))%10;
        y[f[j]]=a[i];
        f[j]--;
    }
    for(i=1;i<=n;i++)
        a[i]=y[i];
}
void sort_radix(int a[], int n, int k)
{
    int i;
    for(i=0;i<=k-1;i++)
        counting(a,n,9,i);
}
```

Listing 7: Sortare pe baza cifrelor

```
void insert_pas(int a[], int n, int h)
{
    int i, j;
    double aux;
    for(i=h+1;i<=n;i++)
    {
        aux=a[i];
        j=i-h;
        while(j>=1 && aux<a[j])
        {
            a[j+h]=a[j];
            j-=h;
        }
        a[j+h]=aux;
    }
}
void sort_shell(int a[], int n)
{
    int h=1;
    while(h<=n)
        h=3*h+1;
    do
    {
        h=(int)h/3;
        insert_pas(a,n,h);
    }
    while(h!=1);
}
```

Listing 8: Algoritmul Shell Sorting

```
void sort_shaker(int a[], int n)
{
    int s=1, d=n, i, t, aux;
    do
    {
        t=0;
        for(i=s;i<=d-1;i++)
            if(a[i]>a[i+1])
            {
                aux=a[i];
                a[i]=a[i+1];
                a[i+1]=aux;
                t=i;
            }
        if(!t)
        {
            d=t;
            t=0;
            for(i=d;i>=s+1;i--)
                if(a[i]<a[i-1])
                {
                    aux=a[i];
                    a[i]=a[i-1];
                    a[i-1]=aux;
                    t=i;
                }
            s=t;
        }
    }
    while(t!=0 && s!=d);
}
```

Listing 9: Shaker Sort

4 Rezultate și analiză

În continuare, vom nota timpii de rulare necesari fiecărui algoritm de sortare amintit, pentru seturi de 1000 de numere, 10000 de numere, 10000 de numere sortate crescător, 10000 de numere sortate descrescător, 1000000 de numere, respectiv 100000000 de numere. Am utilizat seturi de date în care numerele sunt cuprinse între 1 și 1000.

Rezultatele experimentale pot fi observate în Tabela 1.

Observăm că rezultatele experimentale sunt în concordanță cu teoria. Pentru seturi de date relativ mici, de 1000 de elemente sau 10000 de elemente, timpii de execuție diferă în mică măsură de la un algoritm la altul. Diferența

începe să fie semnificativă începând cu seturi de peste 1000000 elemente. În acest caz, timpul de execuție pentru algoritmi precum Merge, Quick, Counting, Radix și Shell Sort este înjumătățit, în comparație cu ceilalți algoritmi. Pentru sortarea a 100000000 de numere trendul se păstrează, diferența dintre timpii de execuție devenind și mai evidentă. Observăm că metodele cele mai intuitive de sortare, respectiv Insertion, Select și Bubble Sort, sunt și cele mai costisitoare din punctul de vedere al timpului de execuție, iar metodele complexe sunt în general mai eficiente în cazul unor seturi mari de date.

Algoritm de sortare	Timp de rulare (s)					
	1000	10000	10000 sortate crescător	10000 sortate descrescător	1000000	100000000
Insertion Sort	0.1	1.049	0.949	1.109	138.686	504.400
Selection Sort	0.301	1.055	1.071	1.117	243.306	nt
Bubble Sort	0.132	1.312	0.954	1.25	186.034	nt
Merge Sort	0.272	0.949	0.942	0.913	61.424	2226.583
Quick Sort	0.158	0.934	0.98	0.947	73.272	5301.821
Counting Sort	0.119	0.939	0.923	0.964	63.8	nt
Radix Sort	0.116	0.953	0.955	0.934	61.024	nt
Shell Sort	0.123	1.174	1.155	1.197	61.557	nt
Shaker Sort	0.143	1.266	1.303	1.317	172.742	nt

Tabela 1: Rezultate experimentale ale timpilor de rulare pentru algoritmii de sortare.

În figura 1 sunt prezentate rezultatele experimentelor pentru diferiți algoritmi de sortare, unde fiecare bară reprezintă timpul de execuție în secunde pentru un algoritm specific. Experimentele au fost efectuate pe un set de date format din un milion de elemente. Analizând graficul, putem observa variația timpului de executare între diferitele algoritmi de sortare, oferind o perspectivă vizuală asupra performanțelor acestora în contextul datelor mari de intrare.

În figura 1, fiecare număr de pe axa orizontală corespunde unui anumit algoritm de sortare. Astfel, algoritmii reprezentați pe axa orizontală sunt, în ordine: sortarea prin inserție, sortarea prin selecție, sortarea prin interschimbarea elementelor vecine, sortarea prin interclasare, sortarea rapidă, sortarea prin numărare, sortarea pe baza cifrelor, Shell Sorting, Shaker Sort.

Analizând rezultatele experimentale, putem observa că algoritmii Merge Sort, Counting Sort, Radix Sort și Shell Sort necesită un spațiu suplimentar de memorie pentru stocarea datelor intermediare sau a unor structuri auxiliare. De exemplu, Merge Sort folosește un vector auxiliar pentru a combina sub-șirurile sortate, Counting Sort utilizează un tablou suplimentar pentru frecvența aparițiilor fiecărui element, iar Radix Sort necesită un spațiu de

memorie pentru a stoca rezultatele intermediare ale sortării pe baza cifrelor. Shell Sort folosește spațiu suplimentar pentru a reorganiza și a sorta sub-tablourile în etapele intermediare ale algoritmului.

Pe de altă parte, algoritmii precum Insertion Sort, Selection Sort și Bubble Sort nu necesită spațiu suplimentar de memorie, deoarece operează direct asupra tabloului de intrare, fără a folosi structuri auxiliare pentru stocarea datelor intermediare.

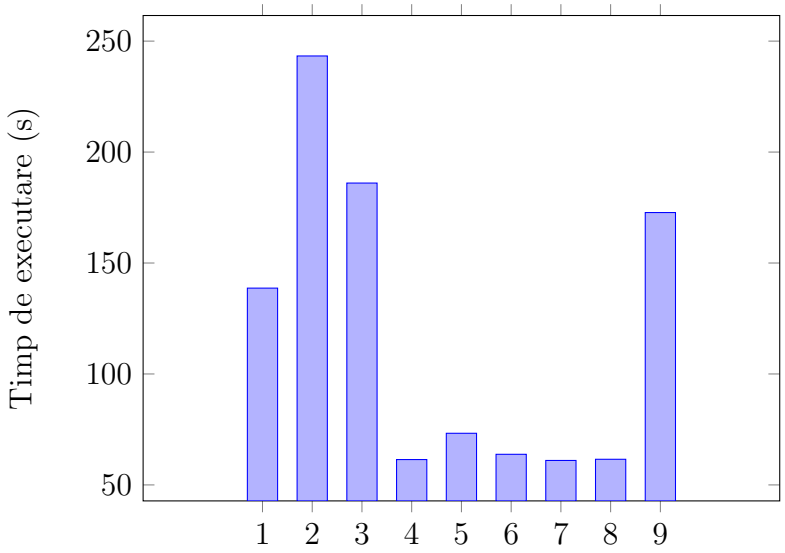

Figura 1: Rezultatele experimentelor pentru diferiți algoritmi de sortare (1,000,000 elemente)

5 Comparația cu literatura

Lucruri similare au fost făcute în [Sedgewick(1990)]. Comparația cu cartea lui Sedgewick arată că rezulatele experimentale obținute în această lucrare sunt similare cu cele obținute de experții ce au abordat problema, însă evidențiază și mici diferențe.

De exemplu, Sedgewick observă că, în cazul unui tablou deja sortat, Quick Sort este unul dintre cei mai nepotriviți algoritmi, în acest caz pretându-se utilizarea unor algoritmi considerați, în general, mai ineficienți, precum Insertion Sort sau Bubble Sort. Această concluzie se poate desprinde și din rezultatele experimentale obținute în studiul de față, timpul de execuție pentru Quick Sort (0.98 s) fiind mai mare decât pentru Insertion (0.949 s) și Bubble Sort (0.954 s).

De asemenea, autorul recomandă folosirea algoritmului Shell Sort, mai ales pentru seturi de date relativ mici, fiindcă este ușor de implementat comparativ cu alți algoritmi și destul de eficient. Totuși, acesta face mențiunea că algoritmii mai complecși, cum ar fi Counting Sort sau Radix Sort au o performanță mai bună în majoritatea cazurilor, dar implementarea lor este mai greoaie. Aceste fapte reies și din lucrarea noastră, observându-se că ultimii doi algoritmi menționați au timpi de execuție mai mici decât Shell Sort pentru cele mai multe seturi de date, însă implementarea prezentată este mai dicilă decât în cazul celorlați algoritmi.

O altă abordare este prezentată în [Cormen and Leiserson(2009)]. Astfel, putem concluziona că la algoritmi precum Insertion și Selection Sort, cazul cel mai favorabil și cazul cel mai defavorabil sunt cele menționate în lucrarea lui Cormen. Putem observa că pentru acești algoritmi timpul de execuție în cazul unui tablou deja sortat crescător este cel mai mic, în timp ce pentru un tablou sortat descrescător timpul de execuție este cel mai mare. În cazul unui tablou nesortat, timpul de execuție se situează între cele două extreme. În cazul Quicksort, întâmpinăm situația opusă, timpul de execuție fiind maxim în cazul unui tablou sortat.

6 Concluzii și direcții viitoare

În acest studiu, am investigat performanța mai multor algoritmi de sortare pe diverse seturi de date, cu scopul de a evalua eficiența lor în diferite contexte. Pe baza rezultatelor obținute, putem trage următoarele concluzii și identifica direcții viitoare de cercetare:

6.1 Concluzii

Performanța algoritmilor de sortare variază semnificativ în funcție de dimensiunea și caracteristicile setului de date utilizat. Algoritmii de sortare eficienți pentru seturi de date mici pot să nu fie la fel de eficienți pentru seturi de date mari și viceversa.

Algoritmul de sortare rapidă a demonstrat o performanță remarcabilă pe seturi de date mari și distribuție aleatoare, având o complexitate a timpului de execuție de O(n log n).

Pentru seturi de date mici și deja parțial sortate, sortarea prin inserție și selecție au oferit rezultate comparabile sau chiar mai bune decât algoritmul de sortare rapidă.

Sortarea prin numărare și sortarea pe baza cifrelor au excelat în sortarea datelor întregi sau cu valori limitate și distribuții uniforme.

Algoritmul de sortare prin shaker a prezentat performanțe modeste, însă poate fi îmbunătățit prin optimizări și ajustări ale implementării.

6.2 Limitări ale studiului

Unul dintre principalele limite ale acestui studiu este dimensiunea și diversitatea seturilor de date utilizate. Extinderea experimentelor la seturi de date mai mari și mai variate ar putea oferi o perspectivă mai cuprinzătoare asupra performanței algoritmilor.

De asemenea, arhitectura hardware utilizată poate influența rezultatele și generalizarea acestora către alte medii de lucru ar trebui făcută cu precauție. Performanța algoritmilor poate fi influențată de calitatea implementării și de nivelul de optimizare a codului. O implementare suboptimală sau lipsa unor optimizări specifice ar putea afecta rezultatele.

6.3 Direcții viitoare

Investigarea și implementarea unor variante optimizate ale algoritmilor de sortare existenți pentru a obține performanțe îmbunătățite pe diverse tipuri de date și platforme hardware.

Studiul impactului arhitecturilor hardware moderne și tehnologiilor de paralelizare asupra performanței algoritmilor de sortare.

Extinderea cercetării pentru a include evaluarea performanței algoritmilor de sortare pe seturi de date din domenii specifice, cum ar fi imagistica medicală sau analiza financiară.

Bibliografie

[Sedgewick(1990)] Robert Sedgewick. *Algorithms in C*. Addison-Wesley Publishing Company, 1990.

[Friend(1956)] Edward H. Friend. Sorting on electronic computer systems. *J. ACM*, 3:134–168, 1956. URL `https://api.semanticscholar.org/CorpusID:16071355`.

[von Neumann(1982)] John von Neumann. *First Draft of a Report on the EDVAC*, pages 383–392. Springer Berlin Heidelberg, Berlin, Heidelberg, 1982. ISBN 978-3-642-61812-3. doi: 10.1007/978-3-642-61812-3_30. URL `https://doi.org/10.1007/978-3-642-61812-3_30`.

[Hoare(1962)] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1): 10–16, 01 1962. ISSN 0010-4620. doi: 10.1093/comjnl/5.1.10. URL `https://doi.org/10.1093/comjnl/5.1.10`.

[Seward(1954)] H. H. Seward. Information sorting in the application of electronic digital computers to business operations. Master's thesis, Digital Computer Laboratory, 1954.

[Shell(1959)] D. L. Shell. A high-speed sorting procedure. *Commun. ACM*, 2(7):30–32, jul 1959. ISSN 0001-0782. doi: 10.1145/368370.368387. URL `https://doi.org/10.1145/368370.368387`.

[Cormen and Leiserson(2009)] Thomas Cormen and Charles Leiserson. *Introduction to Algorithms Third Edition*. The MIT Press, 2009.