Compararea teoretică și experimentală a unor metode de sortare

Humița Ana-Maria*

Realizat: 25.04.2023-12.05.2023

Rezumat

Lucrarea cuprinde o comparație teoretică și experimentală a unor metode cunoscute de sortare.Sortare făcând referire la ordonarea unei mulţimi de elemente, cu scopul de a facilita căutarea ulterioară a unui element dat.

Cuvinte cheie-BubbleSort,Sortare rapidă,Sortare prin selecție,Sortare prin interclasare,Merge Sort,Sortare prin numărare

*Universitatea de Vest din Timișoara, Facultatea de Matematică și Informatică, Departamentul de Informatică, E-mail: ana.humita03@e-uvt.ro

Cuprins

Cuprins

Listă de tabele

Listă de figuri

Fragmente de cod

1 Introducere

În această secţiune vom folosi notaţiile din [1].Majoritatea aplicațiilor care folosesc computerul în zilele noastre necesită ca datele să fie stocate într-un anumit fel.Această procedură care pune datele în ordine se numește sortare și se folosește în diverse câmpuri.Sortarea este folosită pentru organizarea și filtrarea uriașelor cantități de date colectate.În plus, arată o mare semnificație în calculul științific modern și prelucrarea datelor comerciale.
De mulți ani informatica se confrunta cu probleme de calcul în sortare a unei cantitati seminificative de date. Multe aplicații având mari baze de date care necesită algoritmi de sortare mai rapizi și mai buni pentru a da rezultate rezultate optime. Prin urmare, este necesar să existe paralelism crescut astfel încât primitivele de sortare eficiente să poată fi implementate. Performanța bazei de date bazându-se pe tipul de algoritm de sortare utilizat.

1.1 Motivaţie

Sortarea manuala poate fi dificilă și costisitoare din punct de vedere al timpului,mai ales atunci când lucrăm cu liste mari.

Algoritmii de sortare au 2 mari beneficii:

- organizarea datelor mai rapid
- reducerea greșelilor
- În plus algoritmii de sortare au o mare aplicabilitate în practică,în cazul firmelor de exemplu unde se lucrează cu mii de înregistrări,iar algoritmii ne permit să aranjăm datele după anumite criterii cu un singur click.

În lucrarea prezentă considerăm următoarele metode de sortare:

Bubble sort: Se parcurge tabloul de sortat și se compară elementele vecine iar dacă acestea nu se află în ordinea corectă se interschimbă. Parcurgerea se reia până când nu mai e necesară nici o interschimbare.

Sortare rapidă: Se alege un element special al listei, numit pivot,se ordonează elementele listei,astfel încât toate elementele din stanga pivotului să fie mai mici sau egale cu acesta și toate elementele din dreapta pivotului sa fie mai mari sau egale cu acesta.

Sortare prin inserție: Începând cu al doilea element al tablolui v[1...n], fiecare element este inserat pe pozitia adecvata în subtabloul pe care il precede.

Sortare prin interclasare: Un algoritm de tipul divided et impera,unde tabloul este împartit în 2 subtablouri, în care cele două subtablouri sunt sortate in paralel ,pentru ca la final sa fie aduse impreuna in tabloul final.

Sortare prin numarare: Ideea fundamentala a acestui algoritm este de a determina,pentru fiecare element al tabloului de sortat,cate dintre elemente sunt mai mici decat el.Acest lucru este facut prin frecvente cumulate fara a compara elementele intre ele.

Sortare prin selecție: Pentru fiecare poz. i ,incepând cu prima, se selecteaza din subtablou ce începe cu acea poziție cel mai mic element și se amplasează pe locul respectiv

1.2 Instrucțiuni de citire

- Lucrarea cuprinde 6 capitole care au ca și scop descrierea metodelor de sortarea și experimentele realizate pe algoritmii de sortare.
- *Capitolul 1* prezinta o introducere generală a metodelor de sortare, având răspuns la următoarele întrebări după parcurgerea acestuia.

- Unde sunt folosite?Cu ce scop au apărut?Cât de stabile și eficiente sunt?
- *Capitolul 2* descrie amănunțit complexitatea metodelor de sortare,cum ar fi timpul de execuție,cazuri favorabile,nefavorabile.
- *Capitolul 3* cuprinde conceptele de bază ale fiecărei metodelor discutate,pseudocod și fragmente din codul implementat.
- În *capitolul 4* se conturează partea experimentală,timpii de execuție ai fiecărui program .
- *Capitolul 5* cuprinde o comparație cu literatura,avantaje,dezavantaje ai fiecărei metode experimentate,problemele întâlnite.
- *Capitolul 6* cuprinde soluții și direcții viitoare urmat de bibliografie.

1.3 Descrierea informală a soluției

În acest subcapitol vom prezenta proprietățiile solutiilor algoritmilor de sortare în ceea ce privește stabilitate,complexitatea spațială.

Bubble Sort

La metoda Bubble Sort comparăm fiecare pereche adiacentă.
Complexitate spatiala

- Deoarece folosim doar o cantitate constantă de memorie suplimentară, complexitatea spațiului este O(1).

Stabilitate

- Bubble Sort este stabilă cu condiția să se folosească inegalitatea strictă

```
x[j]>x[j+1]
```

Sortare rapidă

La Sortarea rapidă se alege un element special al listei, numit pivot,se ordonează elementele listei după anumite proprietăți.
Complexitate spatiala

- Deși quicksort nu folosește spațiu auxiliar pentru a stoca elemente, este necesar spațiu suplimentar pentru a crea cadre de stivă în apelurile recursive.
- Worst case: O(n)-acest lucru se întâmplă atunci când elementul pivot este cel mai mare sau cel mai mic element
- Best case: O(log n)-acest lucru se întâmplă când pivotul se află pe poziția corectă

Stabilitate

- QuickSort nu este o metoda stabila,deoarece schimbam elementele în funcție de poziția pivotului,iar alegerea pivotului poate fi sensibilă.

Sortare prin inserție

La sortarea prin inserție împarte tabloul în două părți: un tablou de elemente deja sortate și un alt tablou de elemente rămase de sortat.

Complexitate spatiala

- Deoarece folosim doar o cantitate constantă de memorie suplimentară, complexitatea spațiului este O(1).

Stabilitate

- Atâta timp cât key mai mare decât x[j] algoritmul de sortare este stabil,dacă se folosește key este mai mare sau egal cu x[j] stabilitatea nu mai este asigurată.Pentru fiecare comparație efectuată se realizează deplasarea unui element cu o singură poziție.

Sortare prin interclasare

Un algoritm de tipul divided et impera,unde tabloul este împartit în 2 subtablouri.

Complexitate spatiala

- Deoarece folosim un tablou cu dimensiunea de cel mult n pentru a stoca subtablourile îmbinate, complexitatea spațiului este O(n).

Stabilitate

- Sortarea prin interclasare este stabilă,intrucat elementele cu valori egale,apar în aceeași ordine în output.Această sortare este cea mai eficientă pentru linked lists.

Sortare prin numărare

Ideea fundamentala a acestui algoritm este de a determina,pentru fiecare element al tabloului de sortat,cate dintre elemente sunt mai mici decat el.

Complexitate spatiala

- Folosim un tablou de dimensiunea n, astfel încât complexitatea spațiului se dovedește a fi O(n).

Stabilitate

- Counting Sort e o metodă de sortare stabilă si e adesea folosită în algoritmul radix sort,care se bazează pe stabilitatea sa.

Sortare prin selecție

Sortarea prin selecție este un algoritm simplu de sortare care împarte tabloul în două părți: un subgrup de elemente deja sortate și un subgrup de elemente rămase de sortat.

Complexitate spatiala

- Deoarece nu folosim nicio structură de date suplimentară, complexitatea spațiului este O(1).

Stabilitate

- Algoritmul de sortare prin selecție nu este stabil,întrucât minimul este interschimbat cu poziția curentă.Dacă în locul unei singure interschimbări s-ar realiza deplasarea elementelor subtabloului x[i,. . . ,k-1],iar x[k](salvat într-o aux) s-ar transfera în x[i] algoritmul ar deveni stabil.

1.4 Exemple simple ce ilustrează problema si soluția

- Prespunem că avem o listă simplă de numere și dorim să o sortăm folosind **bubble sort** .
- Lista pe care dorim să o sortăm

  5, 2, 8, 4, 1

- Pentru a sorta această listă folosind bubble sort, vom începe la începutul listei și vom compara primele două numere (5 și 2). Deoarece 5 este mai mare decât 2, le vom inversa, rezultând lista: 2, 5, 8, 4, 1. Vom continua să comparam perechi de numere adiacente, inversându-le dacă este necesar, până când ajungem la finalul listei. În acel moment, vom începe din nou de la început și vom repeta procesul până când nu mai sunt necesare alte inversări.
- Prespunem că avem o listă simplă de numere și dorim să o sortăm folosind **sortarea prin insertie** .
- Lista pe care dorim să o sortăm

  5, 2, 8, 4, 1
  Pentru a sorta această listă folosind insertion sort, vom începe cu al doilea număr din listă (2) și îl vom compara cu primul număr din listă (5). Deoarece 2 este mai mic decât 5, îl vom insera înaintea lui 5, rezultând lista: 2, 5, 8, 4, 1. Apoi vom lua al treilea număr din listă (8) și îl vom insera în poziția corectă relativ la primele două numere rezultând lista: 2, 5, 8, 4, 1. Vom continua să inseram fiecare număr ulterior în poziția corectă relativ la numerele sortate anterior până când întreaga listă este sortată.
- Prespunem că avem o listă simplă de numere și dorim să o sortăm folosind **sortarea prin selecție** .

- Lista pe care dorim să o sortăm

  5, 2, 8, 4, 1
  Pentru a sorta această listă folosind selection sort, vom găsi mai întâi cel mai mic număr din listă (care este 1). Apoi, vom inversa acest număr cu primul număr din listă (5), rezultând lista: 1, 2, 8, 4, 5. Vom găsi apoi cel mai mic număr din listă (care este 2) și îl vom inversa cu al doilea număr din listă, rezultând lista: 1, 2, 8, 4, 5. Von continua să găsim cel mai mic număr din listă și să îl inversăm cu următorul număr din listă până când lista este sortată.

2 Prezentare formală a problemei şi soluţiei

În această secţiune vom folosi notaţiile din [2]

2.1 Formalizare a problemei

Vom prezenta o analiză a complexităţii metodelor de sortare menționate în Analiza algoritmilor -Ciprian Bogdan Chirilă.

Sortarea prin inserție

- În cadrul celui de-al i-lea ciclu FOR,numărul Ci al comparațiilor de chei executate în bucla WHILE,depinde de ordinea inițiala a cheilor,fiind:
- Cel putin 1(secvență ordonată)
- Cel mult i-1(secvență ordonată invers)
- În medie i/2.prespunând că permutările celor n chei sunt egal posibile

$$C_{min} = \sum_{i=2}^{n} 1 = n - 1$$

$$C_{max} = \sum_{i=2}^{n} i - 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

$$C_{med} = \frac{C_{min} + C_{max}}{2} = \frac{n^2 + n - 2}{4}$$

- Best case complexity -atunci când șirul e ordonat O(n)

- Worst case complexity -atunci când șirul e ordonat invers

$$O(n^2)$$

Sortare prin selectie

- 0 interschimbări(secvență ordonată)
- Pentru fiecare i se efectuează 3 interchimbări(secvență ordonată invers)

$$T_{min} = 0$$
$$T_{max} = 3\sum_{i=1}^{n-1} 1 = 3(n-1)$$

- Indiferent de aranjarea inițiala a elementelor,numărul de comparații efectuate este

$$T_{med} = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} 1 = \sum_{i=1} n - 1 = n(n-1)$$

- La numărul Ci de atribuiri executate în cadrul ciclului interior WHILE de tip a[j+1]:=a[j] se mai adaugă 3 atribuiri.
- Best case complexity -atunci când șirul e ordonat

$$O(n^2)$$

- Worst case complexity -atunci când șirul e ordonat invers

$$O(n^2)$$

Bubble sort

- Numerele nu trebuie interschimbate,prin urmare numarul de interes-chimbări efectuate este 0,iar numărul de comparații este n-1(secvență ordonată)
- Numerele trebuie schimbate,iar fiecare interschimbare prespune efectu-area a 3 atribuiri

$$T_{min} = n - 1$$
$$T_{max} = 3\sum_{i=2}^{n}\sum_{j=1}^{i-1} 1 = 3\sum_{i=2}^{n} i - 1 = 3\frac{n(n-1)}{2}$$

- Best case complexity -atunci când șirul e ordonat O(n)
- Worst case complexity -atunci când șirul e ordonat invers

$$O(n^2)$$

Sortare rapida

- Cel mai bun caz e atunci când pivotul pivotul e fie cel mai mare element,fie cel mai mic element.
- În cel mai rau caz,pivotul se află la mijloc sau aproape de mijloc

$$T_{min} = n^2$$

$$T_{max} = 0, n = 1$$
$$T_{max} = 2 * T(\frac{n}{2}) + T_m(n), n > 1$$

- Pentru cel mai defarobil caz vom folosi metoda Master,unde k=2(cele 2 subprobleme trebuie rezolvate) m=2(dimensiunea subproblemei este de n/2)

  $$m^d = 2^1 = 2$$

  ,ne aflăm în cazul 2 când

  $$k = m^d$$

  prin urmare avem cazul

  $$O(n^d * log(n)) = O(n * (log(n))$$

  d=1(etapele de divizare și combinare au cost de 1)
- Best case complexity -atunci când șirul e ordonat

$$O(n^2)$$

- Worst case complexity -atunci când șirul e ordonat invers

$$O(n * log(n))$$

Sortare prin interclasare

- Tabloul se înjumătățește în două subtablouri,prin urmare e vorba de o divizare.În acest caz putem folosi metoda Master pentru a afla ordinul de complexitate.

- Pentru interclasare este este necesar un spațiu de memorie suplimentar, de dimensiunea tabloului care se sortează.
- Atunci când avem un singur număr

$$T = 0, n = 1$$

- În celelalte cazuri

$$T_{max} = T(\frac{n}{2}) + T(n - \frac{n}{2}) + T_m(n), n > 1$$

k=2(cele 2 subprobleme trebuie rezolvate) m=2(dimensiunea subproblemei este de n/2)

$$m^d = 2^1 = 2$$

,ne aflăm în cazul 2 când

$$k = m^d$$

prin urmare avem cazul

$$O(n^d * log(n)) = O(n * (log(n))$$

d=1(etapele de divizare și combinare au cost de 1)

- Cazul de complexitate este

$$O(n * log(n))$$

Sortare prin numărare

- În continuare vom analiza complexitatea metodei de sortare rapide,care determină pentru fiecare element de intrare x, numărul de elemente mai mici decât x și folosește aceste informații pentru a plasa elementul x direct în poziția sa în vectorul de ieșire,vezi [**?**].
- Sortarea prin numărare e adesea folosită în algoritmul Radix Sort care se bazează pe stabilitatea sa.
- Best case complexity -liniar,atunci când toate elementele sunt de același grad

$$O(n)$$

- Worst case complexity -atunci când maximul e mult mai mare decât celelalte elemente

$$O(n + k)$$

3 Modelare şi implementare

În această secţiune vom folosi notaţiile din [3]

3.1 Sortare prin inserție

Concept de bază

Începând cu al doilea element al tablolui v[1....n], fiecare element este inserat pe pozitia adecvata în subtabloul pe care il precede.

Pseudocod

- Primul element se prespune a fi sortat,iar al doilea element este memorat separat în variabila cheie.

```
integer i,j,key;
for i->2,n
key-> v[i]
j=i-1
```

- Se compara cheia cu primul element,daca acesta este mai mare,se insereaza in locul cheii.

```
while( p >= 1) and  (v[j]>= key)
v[j+1]-> v[j]
j->j-1
endwhile
```

- In final,cheia va lua valoarea lui j+1 si se va repeta pasul doi (cheia se compara el ramase,daca sunt mai mari,trec in fata) și se returnează tabloul sortat.

```
v[j+1]->key
return v[1.........n]
```

3.2 Sortare prin interclasare

Concept de bază

Un algoritm de tipul divided et impera,unde tabloul este impartit in 2 subtablouri,in care cele doua doua subtablouri sunt sortate in paralel ,pentru ca la final sa fie aduse impreuna in tabloul final.

Pseudocod

- Impărțim tabloul în două subtablouri

```
m=(st+dr)/2  #mijlocul
Interclasare(x{s...m], x[m+1,....d])
i<-s, j<-m+1;
k<-0
while i<=m and j<=d do
k<-k+1
```

- Se compara cheia cu primul element,daca acesta este mai mare,se insereaza in locul cheii.

```
while( p >= 1) and  (v[j]>= key)
v[j+1]-> v[j]
j->j-1
endwhile
```

- Daca elementele care corespund primului subtablou sunt mai mici decat cele din al doilea,trec in vectorul creat t[k]

```
if x[i]<=x[j]
then t[k]<-x[i]
i<-i+1
```

- Altfel,daca elementele din x[j] sunt mai mici

```
else
t[k]<-x[j]
j<-j+1
endif
endwhile
```

- Se transferă eventual elementele rămase

```
while i<=m do
k<-k+1
t[k]<-x[i]
i<-i+1
endwhille
#la fel si pt j
```

```
void merge(int array[], int const left, int const mid,
           int const right)
{
    auto const subArrayOne = mid - left + 1;
    auto const subArrayTwo = right - mid;


    auto *leftArray = new int[subArrayOne],
         *rightArray = new int[subArrayTwo];
    for (auto i = 0; i < subArrayOne; i++)
        leftArray[i] = array[left + i];
    for (auto j = 0; j < subArrayTwo; j++)
        rightArray[j] = array[mid + 1 + j];

    auto indexOfSubArrayOne
        = 0,
        indexOfSubArrayTwo
        = 0;
    int indexOfMergedArray
        = left;

    while (indexOfSubArrayOne < subArrayOne
           && indexOfSubArrayTwo < subArrayTwo) {
        if (leftArray[indexOfSubArrayOne]<= rightArray[indexOfSubArrayTwo])
            {
            array[indexOfMergedArray]= leftArray[indexOfSubArrayOne];
            indexOfSubArrayOne++;
        }
```

```
        else {
            array[indexOfMergedArray]= rightArray[indexOfSubArrayTwo];
            indexOfSubArrayTwo++;
        }
        indexOfMergedArray++;
    }

    while (indexOfSubArrayOne < subArrayOne) {
        array[indexOfMergedArray]
            = leftArray[indexOfSubArrayOne];
        indexOfSubArrayOne++;
        indexOfMergedArray++;
    }

    while (indexOfSubArrayTwo < subArrayTwo) {
        array[indexOfMergedArray]
            = rightArray[indexOfSubArrayTwo];
        indexOfSubArrayTwo++;
        indexOfMergedArray++;
    }
    delete[] leftArray;
    delete[] rightArray;
```

Fragment de cod 1: Funcţia de partiţie pentru Sortare prin interclasare în limbajul C++

```
void mergeSort(int array[], int const begin, int const end)
{
    if (begin >= end)
        return;

    auto mid = begin + (end - begin) / 2;
    mergeSort(array, begin, mid);
    mergeSort(array, mid + 1, end);
    merge(array, begin, mid, end);
}

// Function to print an array
void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}
```

3.3 Sortare prin selecție

Concept de bază

Pentru fiecare poz. i ,incepând cu prima, se selecteaza din subtablou ce începe cu acea poziție cel mai mic element și se amplasează pe locul respectiv.

Pseudocod

- Şelectam primul element ca și minim

```
for i<- 1,n-1 do
k<- i
```

- Çomparam minim cu al doilea element,daca acesta este mai mic decat minim,acesta se transforma in minim .

```
for j<-i+1,n do
if x[k] < j  then k<-j endif
```

- Şe repeta acest pas pana la ultimul element(pana cand se termina for)

```
endfor
```

- Şe repeta acest pas pana la ultimul element(pana cand se termina for)

```
if k!=i
then x[i]<->x[k]
```

- Şe determina valoarea minima din x[i...n] si se interschimba cu x[i],ca în final să se returneze tabloul sortat.

3.4 Sortare rapidă

Concept de bază

Se alege un element special al listei, numit pivot,se ordonează elementele listei,astfel încât toate elementele din stanga pivotului să fie mai mici sau egale cu acesta și toate elementele din dreapta pivotului sa fie mai mari sau egale cu acesta.

Pseudocod

– Se alege o valoare arbitrară care va fi pivotul.

```
Pivot(x[s.........d])
v<-x[d]
```

– Vom seta două puncte,respective la început și la final

```
i<-s-1
j<-d
while i<j do
```

– Se tot mărește i până se ajunge la o valoare mai mare sau egală cu v.Se tot micșorează j până când se ajunge la o valoare mai mică sau egală cu v.

```
repeat i<-i+1 until x[i]>=v
repeat j<-j+1 until x[j]<=v
endfor
```

– Elementul mai mare vine în locul lui j(partea dreaptă),cel mai mic în locul lui i (partea stangă).

– La final se schimba pivotul cu elementul din stanga(atunci cand indexul lui i e mai mare decat indexul lui j)

```
if i<j then x[i]<->x[j]
endwhile
x[i]<->x[d]
```

```
nclude <iostream>
ing namespace std;

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

  int pivot = arr[high]; // pivot
  int i
      = (low- 1);

  for (int j = low; j <= high - 1; j++) {

      if (arr[j] < pivot) {
          i++;
          swap(arr[i], arr[j]);
      }
  }
  swap(arr[i + 1], arr[high]);
  return (i + 1);


id quickSort(int arr[], int low, int high)

  if (low < high) {

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

Fragment de cod 2: Funcţia de partiţie pentru Sortarea rapida limbajul C++

```
    if (low < high) {

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


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


void printArray(int arr[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        cout << arr[i] << " ";
```

3.5 Sortare prin numărare

Concept de bază

Ideea fundamentală a acestui algoritm este de a determina,pentru fiecare element al tabloului de sortat,câte dintre elemente sunt mai mici decât el.Acest lucru este făcut prin frecvențe cumulate fără a compara elementele între ele.

Pseudocod

– Găsim elementul cu val. maximă din șir

```
Maxim(a[],n)
int max=a[0]
for(int i=1;i<n;i++)
if(a[i]>max)
max<-a[i]
endif
return max
endfor
```

– Inițializăm un șir de lungimea max+1 și numărăm aparițiile fiecărui element.

– Reținem suma frecvențelor și începem prin a căuta primului element din șirul initial în șirul de frecvențe(ex daca primul element e 1 ne mutăm în șirul de frecvente în căsuța cu indexul 1).

```
output[n+1]
max<-getMax(a,n)
count[max+1]
for(int i=0;i<=max;++i)
count[i]<-0
endfor
for(int i=0;i<n;i++)
count[a[i]]++
endfor
for(int i=1;i<=max;i++)
count[i]+=count[i-1]]
 for(int i=0;i<n;i++)
 a[i]=output[i]
 endfor
 endfor
```

```c
#include<stdio.h>

int getMax(int a[], int n) {
    int max = a[0];
    for(int i = 1; i<n; i++) {
       if(a[i] > max)
          max = a[i];
    }
    return max;

void countSort(int a[], int n)
{
    int output[n+1];
    int max = getMax(a, n);
    int count[max+1];


   for (int i = 0; i <= max; ++i)
   {
     count[i] = 0;
   }

   for (int i = 0; i < n; i++)
   {
     count[a[i]]++;
   }

    for(int i = 1; i<=max; i++)
       count[i] += count[i-1];

    for(int i = 0; i<n; i++) {
       a[i] = output[i];
    }
}

void printArr(int a[], int n)
{
     int i;
     for (i = 0; i < n; i++)
         printf("%d ", a[i]);
}

int main() {
     int a[] = { 441, 898, 677, 334, 948, 404, 208, 491, 76
     int n = sizeof(a)/sizeof(a[0]);

     countSort(a, n);
     printf("\nAfter sorting array elements are - \n");
```

Fragment de cod 3: Funcţia de partiţie pentru Sortarea prin numărare limbajul C++

4 Studiu de caz / experiment

Rezultatele experimentale pot fi observate în tabelele de mai jos.Codul a fost testat pe 10,100 și 1000 de numere.Sunt luate în considerare 2 cazuri,atunci când șirul este ordonat și atunci când șirul este ordonat invers pentru a face o comparație a timpiilor necesari de execuție.

	Bubble sort	Sortarea prin inserţie	Sortarea rapidă
10	0.0037	0.0033	0.0016
100	0.0868	0.0382	0.0.0185
1000	6.199	2.6388	0.9953

Tabela 1: Rezultate experimentale obţinute pe un șir ordonat invers.

	Bubble sort	Sortare prin inserție	Sortare rapida
10	0.0037	0.0031	0.0016
100	0.0041	0.0039	0.0874
1000	3.0034	0.0335	2.864

Tabela 2: Rezultate experimentale obţinute pe un șir ordonat.

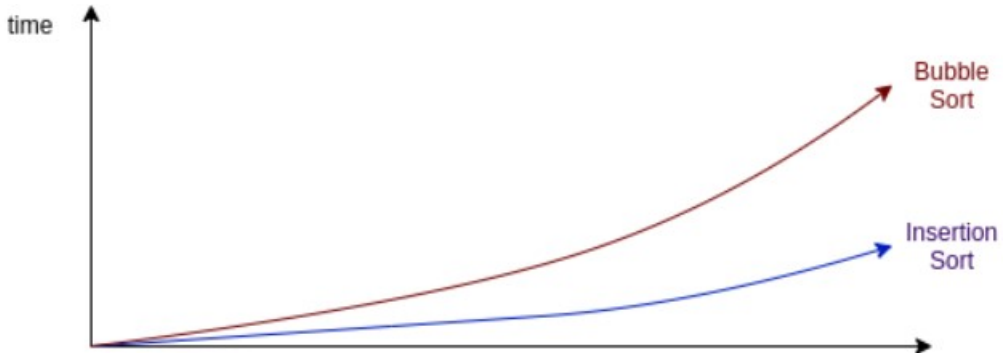
	Interclasare	Selectie	Sortare prin numărare
10	0.0186	0.00041	0.0071
100	0.1482	0.0437	0.0103
1000	1.416	2.0721	0.0329

Tabela 3: Rezultate experimentale obţinute pe un șir ordonat .

	Interclasare	Selecție	Sortare prin numărare
10	0.0262	0.0062	0.0071
100	0.1907	0.0413	0.0133
1000	1.5878	2.9647	0.0336

Tabela 4: Rezultate experimentale obţinute pe un șir ordonat invers.

Fragment de cod 4: Comportamentul metodelor implementate în limbajul C++



Alternativ, putem vizualiza comportamentul implementării noastre în figura de mai jos

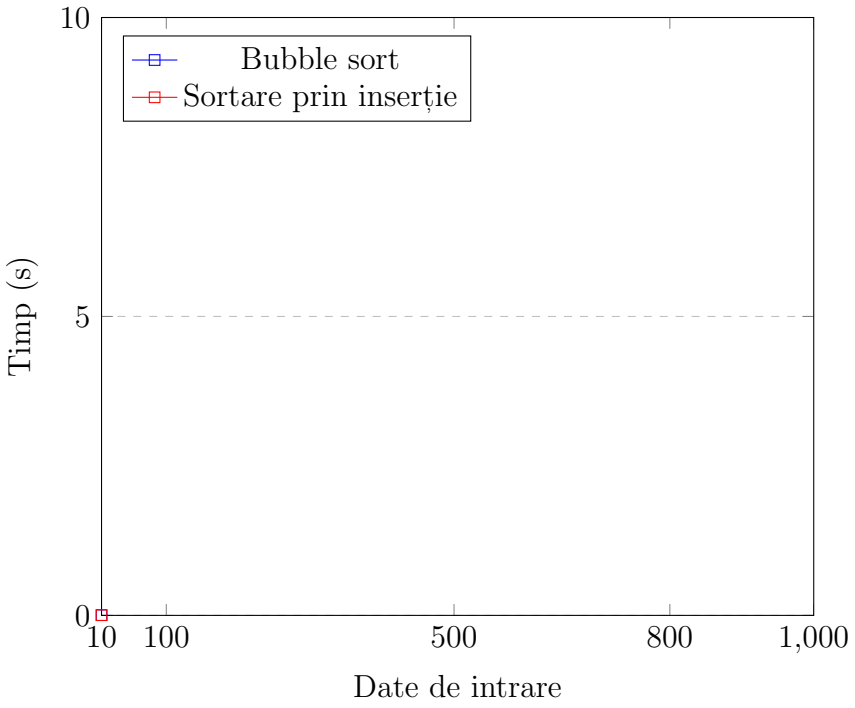


Figura 1: Comportamentul metodelor implementate (grafic liniar).

5 Comparaţia cu literatura

În acest capitol vom prezenta avantajele și dezvantajele metodelor discutate.

Bubble sort: Avantajele principale ale sortării Bubble sort sunt popularitatea și implementare. Principalul dezavantaj al sortării Bubble Sort este faptul că nu se descurcă bine cu o listă care conține un număr mare de articole. Acest lucru se datorează faptului că sortarea cu bule necesită pași de procesare n pătrați pentru fiecare n număr de elemente care urmează să fie sortate. Ca atare, sortarea Bubble Sort este potrivită în mare parte pentru predarea academică, dar nu pentru aplicații din viața reală.

Sortare rapidă: Sortarea rapidă este considerată cel mai bun algoritm de sortare. Acest lucru se datorează avantajului său semnificativ în ceea ce privește eficiența, deoarece este capabilă să facă față bine unei liste uriașe de articole.Dezavantajul sortării rapide este că performanța sa în cel mai rău caz este similară cu performanța medie a sortării cu bule, inserare sau selecție.

Sortare prin inserție: Principalele avantaje al sortării prin inserție sunt simplitatea sa în ceea ce privește implementare și spațiul minim de memorie ocupat.Dezavantajul sortării prin inserție este că nu funcționează la fel de bine ca alți algoritmi de sortare. Cu pași n pătrați necesari pentru fiecare n element care urmează să fie sortat, sortarea prin inserție nu se descurcă bine cu o listă uriașă.

Sortare prin interclasare: O sortare prin interclasare este mai rapidă și mai eficientă decât alte tipuri de sortări atunci când se utilizează liste mai lungi. Cu toate acestea, folosește mai multă memorie și poate dura mai mult pentru a sorta liste mai scurte.

Sortare prin numarare: Sortarea prin numărare funcționează în general mai rapid decât toți algoritmii de sortare bazați pe comparație, cum ar fi sortarea prin interclasare și sortarea rapidă. Cu toate acestea,sortarea prin numărare este ineficientă dacă intervalul de valori care trebuie sortat este foarte mare

Sortare prin selecție: Principalul avantaj al sortării prin selecție este că funcționează bine pe o listă mică. Performanța sa este influențată de ordonarea inițială a articolelor înainte de procesul de sortare. Din acest motiv, sortarea de selecție este potrivită doar pentru o listă de câteva elemente care sunt în ordine aleatorie.

6 Concluzii şi direcţii viitoare

Această lucrare a cuprins o analiză a metodelor de sortare atât din punct de vedere al complexității, stabilității și a comportamentelor.

Pe parcursul lucrarării s-au conturat atât avantajele cât și dezavantajele metodelor propuse.În sectiunea experimentală având posibilitatea de a observa prin intermediul tabelelor și a graficelor cele mai eficiente și ineficiente metode în ceea ce privește timpii de execuție în funcție de șiruri și numărul de date cu care se lucrează.De asemenea, este important de luat în considerare complexitatea algoritmiilor de sortare, atât în termeni de spațiu și stabilitate, deoarece aceastea pot afecta performanțele aplicațiilor noastre.

În concluzie, directiile viitoare în cercetarea metodelor de sortare includ explorarea de noi algoritmi și îmbunătățirea celor existenți pentru a optimiza performanța , eficiența și pentru a accelera procesul de sortare și aplicarea a metodelor de sortare în domenii noi , cum ar fi sortarea datelor genetice sau învățarea automată.

Bibliografie

[1] Minsoo Jeon and Dongseung Kim. Parallel merge sort with load balancing. *International Journal of Parallel Programming*, 31:21–33, 2003.

[2] John L Weatherwax. *Solution Manual for: Introduction to ALGORITHMS by T. Cormen, C. Leiserson, and R. Rivest.* MIT Press Ltd, 2021.

[3] Daniela Zaharie. Influence of crossover on the behavior of differential evolution algorithms. *Applied soft computing*, 9(3):1126–1138, 2009.