Compararea metodelor de sortare în domeniul informaticii
LaTeX

Eric Robert Iov
Departamentul de Informatică
Facultatea de Matematică şi Informatică,
Universitatea de Vest Timişoara, România,
Email: eric.iov03@e-uvt.ro

6 martie 2024

Rezumat

Aceasta este o lucrare experimentală care se ocupă de o problemă destul de cunoscută în informatică, și anume sortarea. Mai exact, această lucrare prezintă compararea diferitelor metode de sortare și analizează avantajele și dezavantajele lor în anumite situații.
În ceea ce privește soluția, lucrarea prezintă o analiză detaliată a unor algoritmi de sortare cunoscuți, precum bubble sort, insertion sort, merge sort și quicksort. Se compară timpul de execuție și eficiența acestora și se formează o concluzie pe baza rezultatelor experimentale.

Cuprins

Listă de tabele

Listă de figuri

1 Introducere

Sortarea unor șiruri de numere sau a unor date reprezintă o problema elementară a programării, fiind printre cele mai des întâlnite situații care trebuie rezolvate. Sortarea datelor reprezintă rearanjarea unor date de același tip, după un anumit criteriu.

1.1 Motivaţie

Problema abordată în această lucrare este importantă pentru că există numeroase aplicații utile în practică. Așa cum am menționat mai sus, problema sortării este o parte esențială a prelucrării datelor. Această problemă nu are o soluție adecvată datorită a două mari categorii în care se împart aceste metode:

- Metodele ușor de înțeles, dar care nu sunt eficiente în cazul unui număr mare de elemente
- Metodele care sunt mai greu de înțeles, de implementat și care consumă mai multă memorie, dar care sunt mult mai eficiente și mai utile în cazul prelucrării unui număr mare de date.

1.2 Descrierea informală a soluției

În lucrarea prezentă considerăm următoarele metode de sortare:

- BubbleSort
- Prin selectie directa
- Prin inserție
- QuickSort
- MergeSort
- CountingSort

Pentru fiecare metodă de sortare se va prezenta pseudocodul, explicația metodei de sortare și a complexității acesteia, și timpul necesar sortării unui anumit număr de elemente, după care se vor compara datele experimentale obținute.

1.3 Exemple simple ce ilustrează problema și soluția

- Program de sortare a unui număr mic de angajați după un anumit criteriu. Inițial folosim o sortare complexă și greu de implementat. După un timp realizăm că programul ar putea să funcționeze la fel de eficient și dacă am folosi o sortare ușoară, dar cu o complexitate mai mare, ceea ce nu ar afecta timpul de execuție, datorită numărului mic de angajați. De asemenea, implementând o sortare de acest gen, reducem consumul de memorie, nefiind nevoie de structuri auxiliare precum în cazul altor sortări.

- Program de sortare a unui număr mare de elemente. Inițial folosim o sortare ușor de implementat pentru a realiza programul. La prima rulare a programului așteptăm un timp destul de mare pentru a se finaliza. Ne gândim ce s-ar întâmpla dacă numărul de elemente ar fi și mai mare, și cât de afectat ar fi timpul de execuție. Astfel, utilizăm o altă sortare mai eficientă pentru a face același lucru, rezultatul fiind unul mult mai bun din punct de vedere al timpului de execuție.

1.4 Exemplu complex

Să presupunem că avem un task ce constă în sortarea unei liste de persoane în ordine alfabetică pentru înregistrarea utilizatorilor unei aplicații. Aplicația, fiind la început, are un număr mic de utilizatori, fapt pentru care folosim o sortare ușor de implementat și de înțeles. Problema apare când numărul de utilizatori crește, iar sortarea devine ineficientă datorită timpului mare necesar pentru a aranja și afișa lista de utilizatori după criteriul propus. Astfel, învățăm o nouă metodă de sortare care este dificil de înțeles și implementat, dar care este mult mai eficientă pentru aplicația noastră. Astfel, am reușit să optimizăm aplicația, iar datele sunt ordonate și prelucrate mult mai rapid, dar am pierdut timp pentru a reuși să învățăm și să implementăm noua metodă de sortare.

1.5 Declarație de originalitate

Contribuții:

- implementarea tuturor sortărilor abordate
- obținerea rezultatelor experimentale

- compararea rezultatelor experimentale
- formularea întregii lucrări pe baza celor de mai sus

1.6 Instrucțiuni de citire

În următoarele secțiuni se vor prezenta:

- ideile care stau la baza fiecărei sortări, pseudocodul acestora și a notațiilor folosite pentru a înțelege fiecare concept
- descrierea limbajului tehnic și tehnicile de implementare
- compararea datelor obținute cu ajutorul unor tabele și grafice

2 Prezentare formală a problemei şi soluţiei

Bubblesort: Se compară fiecare element cu elementul de lângă el, iar dacă sunt îndeplinite condițiile, acestea își vor schimbă poziția. Complexitatea algoritmului Bubble Sort este de $O(n^2)$, deoarece lista trebuie parcursă de n ori și fiecare element trebuie comparat cu celelalte elemente de $n - i$ ori. Cu toate acestea, implementarea poate fi mai eficientă în cazul în care vectorul este aproape sortat, deoarece evită sortarea elementelor care sunt deja în ordine.

```
n = lungimea vectorului v
schimbat = adevărat
repetă
    schimbat = fals
    pentru i de la 0 la n-2:
        dacă v[i] > v[i+1]:
            interschimbă v[i] cu v[i+1]
            schimbat = adevărat
atâta timp cât schimbat
```

Sortarea prin selecție directă: Ideea care stă la baza acestei metode de sortare este următoarea: pentru fiecare poziție , i (începând cu prima) se caută minimul din subtabloul $x[i..n]$ și acesta se interschimbă cu

elementul de pe poziția i. Și acest algoritm are o complexitate $O(n^2)$, deoarece pentru fiecare element, trebuie să căutăm minimul în vectorul nesortat, fapt care implică un ciclu de lungime n și unul de lungime $n - i$.

```
n = lungimea vectorului v
pentru i de la 0 la n - 1
   Minim = v[i]
   poziţieMinim = i
   pentru j de la i + 1 la n
        daca v[j] < Minim
            Minim = v[j]
            poziţieMinim = j
        daca poziţieMinim != i
            schimbă v[i] cu v[poziţieMinim]
```

Sortarea prin inserție: Fiecare element al tabloului, începând cu al doilea, este inserat în subtabloul care îl precede astfel încât acesta să rămână ordonat. În cel mai rău caz, acest algoritm are o complexitate $O(n^2)$, deoarece fiecare element trebuie să se mute pe poziția corectă, vectorul fiind ordonat descrescător de unde ar rezulta $n * (n-1)/2$ comparații.

```
n = lungimea vectorului v
Pentru i de la 0 la n
    j = i-1
    aux = v[i]
    cât timp j>=0 si aux < v[j]
        v[j+1] = v[j]
        j = j - 1
    v[j+1] = aux
```

QuickSort: Algoritmul caută un pivot, astfel încât toate elementele din fața acestuia să fie mai mici decât el, iar toate cele de după el să fie mai mari. Algoritmul are o complexitate, în cel mai bun caz, $O(n * logn)$. În cel mai rău caz, QuickSort poate necesita un timp de $O(n^2)$, care apare atunci când vectorul este deja sortat și pivotul este mereu ales într-unul dintre capetele listei.

```
poziție (limita_stangă,limita_dreaptă)
    i=limita_stangă
    j=limita_dreaptă
    i0=0
    j0=-1
    cât timp i<j
        dacă v[i]>v[j]
            interschimbă v[i] cu v[j]
            aux=i0
            i0=-j0
            j0=-aux
        i=i+i0
        j=j+j0
    returnează i

quick_sort(limita_stangă,limita_dreaptă)
    dacă limita_stangă<limita_dreaptă
        pivot=poziție(limita_stangă,limita_dreaptă)
        quick_sort(limita_stangă,pivot-1)
        quick_sort(pivot+1,limita_dreaptă)
```

MergeSort: Sortarea elementelor cu ajutorul interclasarii. Are la baza tehnica "Divide et impera". Algoritmul are complexitatea $O(n * logn)$ datorita tehnicii de divizare, care împarte problema în alte două subprobleme, care la rândul lor vor fi împărțite în alte două subprobleme de unde rezultă complexitate logaritmică. Acestea vor fi interclasate de unde se deduce complexitatea n.

```
interclasare (lim_st,lim_dr,mij)
    i=lim_st
    j=mij+1
    b[] - un nou vector care să rețină interclasarea celor două subșiruri
    k=0
    cât timp i <= mij si j <= lim_dr
        k=k+1
        dacă v[i] < v[j]
            b[k]=v[i++]
        altfel
```

```
            b[k]=v[j++]
        cât timp i<=mij
            b[++k]=v[i++]
        cât timp j <= lim_dr
            b[++k]=v[j++]
        k=1
        pentru i de la lim_st la lim_dr
            v[i]=b[k++]

    merge_sort (lim_st,lim_dr)
        dacă lim_st+1==lim_dr sau lim_st==lim_dr
            dacă v[lim_st]>v[lim_dr]
                interschimbă v[lim_st] cu v[lim_dr]
        altfel
            mij=(lim_st+lim_dr)/2
            merge_sort(lim_st,mij)
            merge_sort(mij+1,lim_dr)
            interclasare(lim_st,lim_dr,mij)
```

CountingSort: Sortarea numără numărul de apariții al unui element în vector. Numărul de apariții este stocat într-un vector auxiliar, urmând ca elemntele să fie aranjate cu ajutorul acestui vector auxiliar. Acest algoritm are complexitatea $O(n+k)$, unde k este valoarea maximă din vector.

```
n = lungimea vectorului v
counting_sort(v, n)
    max = valoarea_maximă_din v
    inițializam vectorul de frecvență cu 0

    pentru i de la 0 la n - 1
    frecvență[v[i]] = frecvență[v[i]] + 1

    pentru i de la 1 la max
        frecvență[i] = frecvență[i] + frecvență[i - 1]
```

```
pentru i de la n-1 la 0
    vector_sortat[frecvență[v[i]] - 1] = v[i]
    frecvență[v[i]]=frecvență[v[i]] - 1
```

Algoritmii utilizați au fost extrași și adaptați din [1], [4]. De asemenea am folosit notațiile prezentate in [3].

3 Modelarea şi implementarea problemei și soluției

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

#define SIZE 10000000

// Funcția de sortare
void sortare()
{

}


int main() {
    int *v;
    int i;
    clock_t start, end;
    double cpu_time_used;
    srand(time(NULL));

    // Alocarea dinamică a vectorului cu numere aleatoare
    v = (int*)malloc(sizeof(int) * SIZE);
    if (v == NULL) {
        printf("Eroare la alocarea dinamică a memoriei");
        exit(1);
    }
```

```
    for (i = 0; i < SIZE; i++) {
        v[i] = rand() % 1000;
    }

    // Sortarea vectorului și măsurarea timpului de execuție
    start = clock();
    sortare();
    end = clock();

    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;

    printf("Timp pentru %d elemente: %f secunde.\n", SIZE, cpu_time_used);

    // Eliberarea memoriei alocate dinamic
    free(v);

    return 0;
}
```

"Manualul" de sistem: Codul de sus implementează un program în limbajul de programare C, care nu utilizează structuri de date complexe sau aparatură hardware specială. În funcția main se alocă dinamic memoria pentru vectorul de numere întregi, se generează vectorul folosind numere aleatorii iar apoi acestea sunt sortate. Timpul de execuție al funcției este măsurat cu ajutorul funcției clock. Implementarea alocării dinamice de memorie se face cu ajutorul funcției malloc, și care returnează un pointer către o zonă de memorie alocată. După utilizarea vectorului, memoria este eliberată cu ajutorul funcției free.

"Manualul" de utilizare: Pentru a utiliza implementarea de cod de sus, trebuie să compilați codul sursă într-un fișier executabil, care poate fi apoi rulat într-un editor de cod precum "CodeBlocks" sau "Visual Studio Code". Valoarea "SIZE"-ului poate fi modificată pentru a vedea cum se comportă sortarile pentru diferite dimensiuni ale unui vector. De asemenea în subprogramul "sortare" se poate introduce oricare dintre pseudocodurile prezentate mai sus descrise în limbajul C pentru a putea fi compilate. După aceasta, se rulează programul iar timpul este afișat pe ecran.

4 Studiu de caz / experiment

nr. elemente	BubbleSort(s)	SelectionSort(s)	InsertionSort(s)
100	foarte mic	foarte mic	foarte mic
1000	0.001	0.008	0.007
10000	0.19	0.098	0.075
100000	22.01	8.94	5.37
1000000	inf	inf	inf

Tabela 1: Comparație între metodele de sortare mai puțin eficiente

nr. elemente	QuickSort(s)	MergeSort(s)	CountingSort(s)
100	foarte mic	foarte mic	foarte mic
1000	foarte mic	foarte mic	foarte mic
10000	0.001	0.001	0.001
100000	0.006	0.014	0.002
1000000	0.086	0.235	0.004
10000000	0.755	2.411	0.035
100000000	8.468	8.959	0.349

Tabela 2: Comparație între metodele de sortare eficiente

Datele obținute prin repetarea experimentului, care a implicat implementarea diferitelor metode de sortare și generarea datelor de intrare prin intermediul funcției rand(), au furnizat informații relevante în ceea ce privește performanțele si limitările acestor algoritmi. Rezultatele obținute au confirmat faptul că BubbleSort este cea mai lentă metodă de sortare (Tabela 1), în timp ce QuickSort este cea mai rapidă, deși timpul său de execuție este similar cu cel al MergeSort (Tabela 2). În plus, s-a observat că, CountingSort este cel mai rapid în cazul în care elementul maxim a fost unul destul de mic. De asemenea, s-a constatat că, în unele cazuri, comportamentul rezultatelor a fost neașteptat, deoarece la primele rulari ale programului, timpii de sortare au fost mai ridicați, dar au scăzut în rulările ulterioare.
Cu ajutorul celor 3 grafice putem vizualiza si compara grafic modificarea timpului de sortare, odată cu creșterea numărului de elemente.

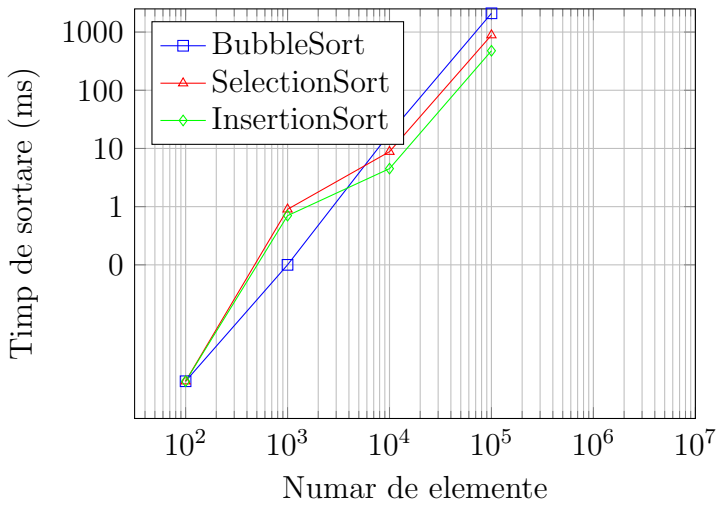

Figura 1: Timpul de sortare pentru BubbleSort, SelectionSort și InsertionSort reprezentat grafic

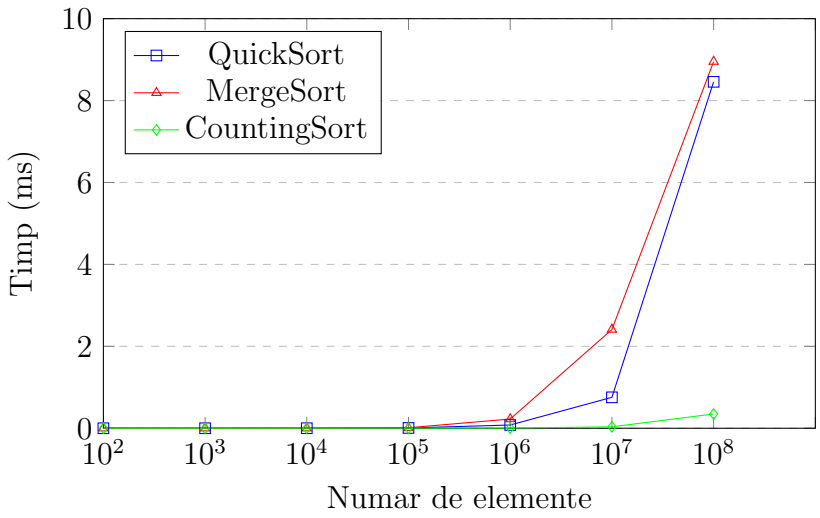

Figura 2: Timpul de sortare pentru QuickSort, MergeSort și CountingSort reprezentat grafic

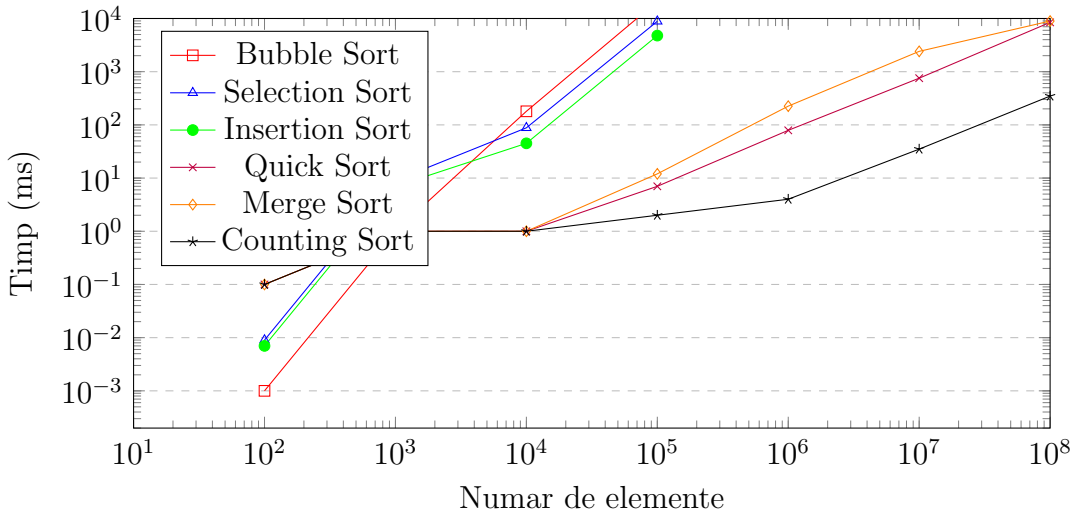

Figura 3: Comparație grafică între toate metodele prezentate

5 Comparaţia cu literatura

O altă abordare este prezentată în "The Art of Computer Programming" scrisa de Donald Knuth [2]. Una dintre principalele diferențe între cele două abordări este că Donald Knuth se concentrează pe analiza matematică și teoretică a algoritmilor de sortare, în timp ce experimentul realizat anterior se concentrează pe performanța practică a acestora. Un avantaj al abordării lui Knuth este acela că in cărțile sale se oferă o prezentare completă a practicilor pentru programarea eficientă. Un avantaj al experimentului descris este că a oferit o evaluare empirică a performanțelor diferitelor algoritmi de sortare utilizând date concrete de intrare. Un dezavantaj al abordării lui Knuth este că teoria poate fi complexă și greu de înțeles, pe când, experimentând cu date concrete și prezentând ideile de baza, se poate înțelege mai eficient. Un dezavantaj al experimentului descris este acela că nu au fost folosite toate datele de intrare posibile si au fost omise scenarii. În ambele lucrări, se subliniază faptul că BubbleSort este cea mai lentă metodă de sortare, în timp ce QuickSort este cea mai rapidă și eficientă metodă, deși timpul său de execuție poate fi similar cu cel al altor metode, cum ar fi MergeSort. În plus, CountingSort este cel mai rapid atunci când elementul maxim este unul destul de mic.

6 Concluzii şi direcţii viitoare

Concluziile principale ale comparării algoritmilor de sortare sunt următoarele:

- Viteza algoritmilor de sortare depinde de câțiva factori, cum ar fi cât de multe date trebuie sortate și cum este implementat algoritmul.
- Dacă avem mai multe date de sortat, atunci vom avea nevoie de un algoritm care să poată face acest lucru rapid și eficient.
- Se poate îmbunătăți performanța fiecărui algoritm de sortare prin optimizare. Asta înseamnă că putem să găsim căi mai rapide sau mai eficiente pentru a-l implementa și astfel să facem ca algoritmul să sorteze datele mai repede.

În viitor, direcțiile de cercetare referitoare la compararea algoritmilor de sortare ar putea include:

- Dezvoltarea și îmbunătățirea algoritmilor deja existenți
- Dezvoltarea tehnologiilor de memorie și procesare poate duc la optimizări ale algoritmilor de sortare existenți sau la apariția unor noi metode de sortare.

Compararea între metodele de sortare este un subiect important în informatică și programare, deoarece performanța acestor algoritmi poate avea un impact semnificativ asupra vitezei și eficienței programelor. În esență, când trebuie să sortăm un număr mare de date, trebuie să alegem cel mai bun algoritm pentru a face acest lucru cât mai rapid și mai eficient posibil.

Bibliografie

[1] Jon L Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. In *Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 360–369, 1997.

[2] E Knuth Donald et al. The art of computer programming. *Sorting and searching*, 3:426–458, 1999.

[3] Ani Kristo, Kapil Vaidya, Ugur Çetintemel, Sanchit Misra, and Tim Kraska. The case for a learned sorting algorithm. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1001–1016, 2020.

[4] Marius MĂGUREAN, Colegiul Militar Liceal, Ștefan cel Mare, and Câmpulung Moldovenesc. Analiza comparativă a metodelor de sortare a vectorilor. *CERINȚE ESENȚIALE ALE EDUCAȚIEI ÎN ÎNVĂȚĂMÂNTUL PREUNIVERSITAR*, page 175.