Grand Prix al Algoritmilor de Sortare. Cursa Contra Timpului

Rebeca Detari
Departamentul de Informatică
Facultatea de Matematică și Informatică
Universitatea de Vest din Timișoara, România
Email: rebeca.detari04@e-uvt.ro

May 2024

Rezumat

Această lucrare propune realizarea unor comparații teoretice și experimentale între anumite metode de sortare, pentru a analiza dacă proprietățile teoretice ale algoritmilor rămân valabile pentru diferite seturi de date, sau dacă există modificări considerabile. În realizarea analizei vom utiliza sortarea în ordine crescătoare. Cele 5 sortări analizate sunt: sortarea cu bule (Bubble sort), sortarea prin inserție, sortarea prin selecție, sortarea prin interclasare (Merge sort) și sortarea rapidă (Quick sort).

După testarea fiecărui algoritm pentru mai multe seturi de date vom compara rezultatele cu ipoteza teoretică, pentru a verifica dacă aceasta rămâne validă. Dacă observăm că rezultatele obținute nu confirmă în întregime ipoteza, vom căuta explicații pentru acest fenomen.

Cuprins

1 Introducere

Algoritmii de sortare sunt esențiali în informatică pentru a organiza datele într-o ordine specifică, pentru a permite accesul și manipularea lor eficientă. Sortarea datelor înainte de a efectua operații de căutare sau acces poate îmbunătăți semnificativ eficiența acestor operații. Algoritmii de sortare sunt adesea folosiți ca o componentă în implementarea altor algoritmi și structuri de date. De exemplu, mulți algoritmi de căutare sau algoritmi de grafuri pot necesita datele să fie sortate înainte de a fi procesate eficient. Din acest motiv analiza eficienței algoritmilor pe diverse seturi de date devine esențială pentru a selecta algoritmul optim pentru o anumită problemă.

1.1 Motivaţie

Având în vedere diversitatea aplicațiilor informatice și nevoia constantă de a găsi soluții eficiente pentru manipularea și procesarea datelor, este esențial să evaluăm și comparăm performanța a diferiți algoritmi de sortare pe seturi de date diferite. Pentru seturi de date mici, variațiile în performanța algoritmilor de sortare pot părea neglijabile sau inexistente. Cu toate acestea, pe măsură ce dimensiunea setului de date crește, alegerea corectă a algoritmului de sortare devine crucială. Timpul necesar pentru sortarea datelor poate influența în mod semnificativ eficiența unei aplicații, având un impact direct asupra experienței utilizatorului și a performanței generale a sistemului.

Declarație de originalitate
Prin prezenta, declar pe propria răspundere că lucrarea de față, intitulată "Grand Prix al Algoritmilor de Sortare. Cursa Contra Timpului", este rezultatul propriului efort intelectual și reprezintă o contribuție originală la domeniul studiat. Textul și ideile prezentate în lucrare sunt rezultatul analizei și interpretării proprii, și nu sunt copiate sau preluate în mod neautorizat din alte surse. În cazul în care am împrumutat idei din alte surse, am asigurat că acestea sunt citate și referite conform standardelor academice.

2 Prezentare formală a problemei şi soluţiei

2.1 Problema:

Fie dat un set de date, reprezentând diverse informații, care trebuie să fie ordonate într-o anumită ordine specifică. Obiectivul este să identificăm și să implementăm algoritmul de sortare optim care să asigure eficiența maximă a

procesului de sortare, în funcție de dimensiunea și caracteristicile setului de date.

2.2 Soluția propusă:

Soluția propusă constă în analiza și evaluarea mai multor algoritmi de sortare, inclusiv Quick sort, Merge sort, Bubble sort, Insertion sort și Selection sort, pentru a determina cel mai eficient algoritm într-o varietate de scenarii. Această evaluare se va face atât din perspectivă teoretică, prin analiza complexității timpului și spațiului, cât și experimentală, prin testarea performanței algoritmilor pe seturi reale de date. Scopul final este de a identifica și de a recomanda algoritmul de sortare optim pentru fiecare tip de set de date, pentru a asigura performanța maximă a procesului de sortare în diferite contexte.

3 Modelare şi implementare

3.1 Sortarea cu bule

Descrierea algoritmului
Tabloul este parcurs de la stânga spre dreapta și elementele adiacente sunt comparate. Dacă nu sunt în ordinea dorită se interschimbă. Procesul este repetat până când tabloul este ordonat.
Clasa de eficiență (complexitate): $T(n) \in O(n^2)$. [2]

Algoritmul utilizat

```
void Sortare_cu_bule(int v[], int n)
{
    for(int i=n-1; i>=1; i--)
        for(int j=0; j<=i-1; j++)
            if(v[j]<v[j+1])
            {
                int aux = v[j];
                v[j] = v[j+1];
                v[j+1] = aux;
            }
}
```

3.2 Sortarea prin inserție

Descrierea algoritmului
Fiecare element al tabloului, începând cu al doilea, este inserat adecvat în subtabloul care îl precede astfel încât acesta (subtabloul care îl precede) să rămână ordonat. [3]
Clasa de eficiență (complexitate): $T(n) \in O(n^2)$; $T(n) \in \Omega(n)$. În cel mai rău caz, când elementele tabloului sunt ordonate invers, clasa de eficiență va fi $T(n) \in O(n^2)$. În cel mai bun caz, când elementele sunt deja sortate în ordinea dorită, clasa de eficiență va fi $T(n) \in \Omega(n)$.

Algoritmul utilizat

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

3.3 Sortarea prin selecție

Descrierea algoritmului
Pentru fiecare poziție i (începând cu prima) se caută minimul din subtabloul v[i..n-1] și acesta se interschimbă cu elementul de pe poziția i. [3] Clasa de eficiență (complexitate): $T(n) \in \Theta(n^2)$.

Algoritmul utilizat

```
void Sortare_selectie(int v[], int n)
{
    for(int i=0; i<n-1; i++)
    {
        int k = i;
```

```
        for(int j=i+1; j<n; j++)
            if(v[k]>v[j])
            k = j;
        if(k!=i)
        {
            int aux = v[i];
            v[i] = v[k];
            v[k] = aux;
        }
    }
}
```

3.4 Sortarea prin interclasare

Descrierea algoritmului

Tabloul v[0..n-1] este împărțit în două subtablouri v[0..(n-1)/2] și v[(n-1)/2..n-1]. Elementele subtablourilor sunt sortate, după care elementele celor 2 subtablouri sunt interclasate și astfel se construiește tabloul sortat.
Clasa de eficiență (complexitate): $T(n) \in \Theta(log(n))$. [4]
Nu este eficient din punct de vedere al memoriei pentru a sorta un număr mare de elemente, deoarece are nevoie de un tablou de aceeași dimensiune cu cel sortat (sau 2 tablouri ale căror dimensiuni adunate dau dimensiunea tabloului inițial), pentru a stoca elementele interclasate la fiecare pas.

Algoritmul utilizat

```
void interclasare(int v[], int l, int m, int r)
{
        int i, j, k;
        int n1 = m - l + 1;
        int n2 = r - m;
        int L[n1], R[n2];
        for (i = 0; i < n1; i++)
            L[i] = v[l + i];
        for (j = 0; j < n2; j++)
            R[j] = v[m + 1 + j];
        i = 0;
        j = 0;
        k = l;
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
```

```
                v[k] = L[i];
                i++;
            }
            else {
                v[k] = R[j];
                j++;
            }
            k++;
        }
        while (i < n1) {
            v[k] = L[i];
            i++;
            k++;
        }
        while (j < n2) {
            v[k] = R[j];
            j++;
            k++;
        }
}
void Sortare_interclasare(int v[], int l, int r)
{
        if (l < r)
    {
            int m = l + (r - l) / 2;
            Sortare_interclasare(v, l, m);
            Sortare_interclasare(v, m + 1, r);
            interclasare(v, l, m, r);
        }
}
```

3.5 Sortarea rapidă

Descrierea algoritmului

Algoritmul sortează o secvență a tabloului alegând un element special al listei, numit pivot. Se ordonează elementele listei, astfel încât toate elementele din stânga pivotului să fie mai mici sau egale cu acesta, și toate elementele din dreapta pivotului să fie mai mari sau egale cu acesta. Se continuă recursiv cu secvența din stânga pivotului și cu cea din dreapta lui.

Clasa de eficiență (complexitate): $T(n) \in O(nlog(n))$. În cazuri defavora-

bile complexitatea algoritmului devine: $T(n) \in O(n^2)$. [4] Asemenea cazuri defavorabile sunt situațiile când elementele sunt ordonate crescător sau descrescător de la început. Anumite metode de alegere a pivotului pot evita acest dezavantaj.

Algoritmul utilizat

```
int partitionare(int v[], int prim, int ultim)
{
    int pivot = v[prim];
    int i = prim;
    int j = ultim;

    while (i < j) {
        while (v[i] <= pivot && i <= ultim - 1)
            i++;
        while (v[j] > pivot && j >= prim + 1)
            j--;
        if (i < j) {
            int aux = v[i];
            v[i] = v[j];
            v[j] = aux;
        }
    }

    int aux = v[prim];
    v[prim] = v[j];
    v[j] = aux;

    return j;
}

void Sortare_Rapida(int v[], int prim, int ultim)
{
        if (prim < ultim)
  {
    int indexpartitionare = partitionare(v, prim, ultim);
    Sortare_Rapida(v, prim, indexpartitionare - 1);
    Sortare_Rapida(v, indexpartitionare + 1, ultim);
  }
}
```

4 Studiu de caz / experiment

În ceea ce urmează vom prezenta sub formă de tabele, rezultatele celor 5 algoritmi de sortare pe 4 tipuri de date pentru: 1.000, 10.000 de elemente și 2 tipuri de date pentru: 100.000, 1.000.000 elemente. Unitatea de măsură utilizată la prezentarea rezultatelor este secunda.

Precizări
Pentru cazurile în care algoritmii au fost aplicați pe elemente deja sortate sau sortate invers trebuie considerată o marjă de eroare creată de faptul că elementele au fost citite și nu generate direct în program. Un alt aspect ce influențează timpul de execuție este dispozitivul utilizat la realizarea acestui experiment sau posibila utilizare a altor aplicații în același timp, ce pot încetini execuția programului.

Pentru a nu încărca tabelele am folosit prescurtări pentru tipurile de date utilizate. Prescurtările utilizate și însemnătatea acestora sunt după cum urmează:
NGR = Numere generate random
LOC = Lista ordonată crescător
LOD = Lista ordonată descrescător
LV1-5R = Listă cu valori cuprinse între 1 și 5 generate random

4.1 Sortarea cu bule

Rezultatele experimentale pentru sortarea cu bule pot fi observate în Tabela 1.

	NGR	LOC	LOD	LV1-5R
1000	0.110	0.562	0.486	0.097
10000	0.521	0.838	0.848	0.479
100000	18.368	-	-	17.509
1000000	3508.266	-	-	1854.655

Tabela 1:

Pentru 1.000.000.000 de elemente estimăm că ar dura cel puțin o zi, concluzie trasă în urma comparării cu rezultatele sortării prin interclasare. 1.000.000 de elemente au fost sortate în aproape 59 de minute prin sortarea cu bule, în timp ce cu sortarea prin interclasare au fost sortate în 20 de secunde. 1.000.000.000 de elemente au fost sortate cu sortarea prin interclasare în 5 ore și 25 de minute, de unde concluzia că utilizând sortarea cu bule timpul necesar sortării ar fi mai mare de 1 zi.

4.2 Sortarea prin inserție

Rezultatele experimentale pentru sortarea prin inserție pot fi observate în Tabela 2.

	NGR	LOC	LOD	LV1-5R
1000	0.100	0.705	0.728	0.107
10000	0.366	0.980	0.981	0.361
100000	3.129	-	-	4.399
1000000	355.863/280.462	-	-	225.472

Tabela 2:

Observăm că pentru 1.000.000 de elemente avem 2 rezultate în cadrul sortării ce utilizează numere generate random. Astfel putem deduce că în funcție de numerele generate, timpul necesar sortării variază, deci de aici rezultă importanța setului de date utilizate.

4.3 Sortarea prin selecție

Rezultatele experimentale pentru sortarea prin selecție pot fi observate în Tabela 3.

	NGR	LOC	LOD	LV1-5R
1000	0.092	0.487	0.593	0.138
10000	0.455	0.790	0.879	0.411
100000	4.361	-	-	4.396
1000000	530.632/411.006	-	-	477.887

Tabela 3:

La fel ca și în cadrul sortării prin inserție, observăm variații ale timpului necesar sortării pentru 1.000.000 de numere generate random.

4.4 Sortarea prin interclasare

Rezultatele experimentale pentru sortarea prin interclasare pot fi observate în Tabela 4.

Pentru 1.000.000.000 de elemente cu valori cuprinse între 1 și 5 pentru care s-a utilizat generarea random timpul necesar sortării utilizând sortarea prin interclasare este de 18934.329 secunde = 5 ore și 25 de minute.

	NGR	LOC	LOD	LV1-5R
1000	0.116	0.425	0.551	0.105
10000	0.328	0.851	0.835	0.365
100000	0.092	-	-	0.090
1000000	20.859	-	-	20.405

Tabela 4:

4.5 Sortarea rapidă

Rezultatele experimentale pentru sortarea rapidă pot fi observate în Tabela 5.

	NGR	LOC	LOD	LV1-5R
1000	0.125	1.917	1.927	0.089
10000	0.316	2	2.079	0.417
100000	2.501	-	-	2
1000000	20.448	-	-	79.690

Tabela 5:

Pentru 1.000.000.000 de elemente cu valori cuprinse între 1 și 5 generate random, timpul necesar sortării prin sortarea rapidă a fost de: 5629.385 secunde = 1 oră și 56 de minute.

Pentru 1.000.000.000 de elemente, numere generate random, rularea programului a durat mai mult de 4 ore. Din acest motiv am considerat că setul generat nu a fost favorabil pentru testare și am întrerupt procesul. În consecință, s-a constatat că, în loc să se încadreze în complexitatea așteptată de: $T(n) \in O(nlog(n))$, algoritmul a prezentat complexitatea asociată cazului defavorabil: $T(n) \in O(n^2)$.

5 Comparaţia cu literatura

Principala comparație cu literatura pe care o vom realiza este cea a sortării rapide, deoarece în cazul acesteia am găsit probabil cea mai interesantă variție a timpului de executare a sortării, raportată la datele sortate.

În „Introduction to algorithms” [1], în capitolul 7, găsim aceeași concluzie pe care am tras-o în urma experimentului efectuat asupra algoritmului de sortare rapidă. Timpul de execuție depinde de partiționare, dacă aceasta este echilibrată sau dezechilibrată. Acest lucru depinde la rândul său de elementele utilizate pentru partitionare. Dacă partiționarea este echilibrată, algoritmul rulează la fel de rapid ca și sortarea prin interclasare. Însă, dacă

partiționarea este dezechilibrată, acesta poate rula la fel de încet ca și sortarea prin inserție. Acest lucru este evident influențat de setul de date și de modul în care este ales pivotul. Atunci când rulăm sortarea rapidă pe un tablou de date generate random, este puțin probabil ca partiționarea să se întâmple întotdeauna în aceeași modalitate la fiecare nivel. Ne așteptăm ca unele împărțiri să fie destul de echilibrate, iar altele să fie destul de dezechilibrate.

În „Sorting & Searching" [2], capitolul 5, observăm o abordare similara a unui asemenea experiment, în care pentru un anumit număr de elemente se iau diferite tipuri de date, asupra cărora se aplică un algoritm de sortare și apoi se constată câte mutări au fost efectuate pentru ca acest set de date să fie sortat. Numărul de mutări efectuate influențează timpul necesar realizării sortării. Acest experiment este efectuat pentru colecții de elemente de dimensiuni diferite (exemplu: n=8, n=100), pentru a putea urmări fluctuarea numărului de mișcări necesare pentru sortare și impactul acestei valori asupra timpului de execuție.

6 Concluzii şi direcţii viitoare

Timpul de execuție variază considerabil în funcție de datele furnizate ce urmează să fie sortate. Cel mai bun exemplu ce susține această afirmație este sortarea a 1.000.000.000 de elemente utilizând sortarea rapidă. Elementele generate random cu valori cuprinse între 1 și 5 au fost sortate în 1 oră și 56 de minute în timp ce pentru elementele generate random cu valori cuprinse între 1 și 1000 sortarea nu s-au terminat în 4 ore, timp după care am oprit rularea algoritmului. Acest fapt este cauzat de alegerea pivotului care este strâns legată de setul de date utilizat și astfel o alegere defavorabilă duce la un caz defavorabil și o complexitate de:$T(n) \in O(n^2)$ în loc de:$T(n) \in O(nlog(n))$.

Am putut trage această concluzie și din datele colectate în urma sortării a 1.000.000 de elemente cu ajutorul sortării prin inserție și a sortării prin selecție. Putem observa că timpul necesar sortării este diferit de la o rulare la alta pentru numerele generate random, deoarece ordinea în care numerele sunt furnizate inițial algoritmlui au un impact considerabil asupra modului în care vor fi mutate în timpul sortării pentru a ajunge într-un final sortate.

Consider relevantă realizarea unui asemenea experiment în procesul de învățare și studiere al algoritmilor de sortare, pentru a realiza diferențele considerabile între timpul de executare al algoritmilor și pentru a înțelege cât de mult este acesta influențat de setul de date utilizat. Observăm că pentru 1000 sau 10.000 de elemente diferențele sunt practic inexistente, fiind mai mici de 1 secundă, aspect ce nu afectează eficiența. Pe măsură ce dimensiunea

valorilor sortate crește, putem observa discrepanțele și astfel conștientiza diferențele dintre algoritmi și prin urmare importanța alegerii algoritmului potivit pentru setul de date pe care dorim să-l sortăm.

Ca o sugestie viitoare, acest experiment ar putea fi realizat pe și mai multe metode de sortare, și tipuri de date diverse. Aplicarea algoritmilor pe un set de date exact, care să fie utilizat pe fiecare aloritm ar putea evidenția și mai bine discrepanțele dintre timpul necesar sortării prin diferite metode. Ar fi interesant ca acest set de date să fie modificat doar într-un loc, o valoare să fie schimbată, pentru a descoperi dacă acest aspect ar crea o diferență considerabilă. Consider o sugestie incitantă realizarea experimentului propus mai sus pe diferite dispozitive, pentru a verifica diferențele create de performanța acestora. De asemenea dispozitivul ar putea fi pus să realizeze diferite alte activități între timp, cu scopul de a vizualiza impactul acestui aspect asupra timpului necesar realizării sortării.

Bibliografie

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

[2] Donald E Knuth. *Sorting and Searching*. Addison-Wesley, 1998.

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

[4] Steven S. Skiena. *The Algorithm Design Manual*. Springer, 2008.