Compararea Teoretică și Experimentală a celor mai cunoscuți Algoritmi de Sortare

Radu Ciobanu

Universitatea de Vest,
Facultatea de Matematică și Informatică,
Timișoara, România
Email: radu.ciobanu02@e-uvt.ro

30 Aprilie 2022

Rezumat. În informatică, pentru aranjarea într-o anumită ordine a unor elemente după un anumit criteriu, sunt necesari niște algoritmi de sortare. Există o multitudine de algoritmi, unii mai eficienți decat alții, dar nu există încă unul perfect. În această lucrare științifică, vom parcurge teoretic si experimental cei mai cunoscuți algoritmi de sortare, dezvăluind mecanicile din spatele fiecăruia; vom hotărî care este mai bun pentru anumite seturi de date, unde și cand ar trebui folosite pentru îndeplinirea scopurilor cu o eficiență semnificativă. După crearea unui program în limbajul C++, am experimentat și notat, pentru patru algoritmi (Quick Sort, Bubble Sort, Merge Sort și Insertion Sort), performanța lor în funcție de timpul de execuție si consumul de energie al procesorului.

Cuvinte-cheie: Algoritmi de Sortare, Eficiență, Complexitate, Timp de execuție, Quick Sort, Bubble Sort, Merge Sort, Insertion Sort

Cuprins

1 Introducere

Sortarea obiectelor este o acțiune întalnită peste tot în lumea înconjurătoare, fie că ne referim la lucruri complexe precum bazele de date ale unor companii, fie la lucruri simple din viața de zi cu zi, aranjamentul obiectelor casnice, dicționare sau pozițiile sociale. Umanitatea a evoluat odată cu această metodă de a face orice mai eficient când vine vorba de informații de dimensiuni mari. În lumea modernă, orice căutare sau modificare în bazele de date ale internetului ar dura extrem de mult timp pentru a fi efectuată dacă nu ar avea la bază un sistem informatic bun, care depinde de o anumită metodă sortare. Spre exemplu, dacă am avea o listă cu numele tuturor persoanelor de pe Pământ, ar dura în cel mai rău caz 7,9 miliarde de căutări pentru a găsi numele persoanei respective. Totuși, dacă numele persoanelor ar fi sortate alfabetic, nu ar mai fi necesară o parcurge a tuturor numelor, ci am începe căutarea din punctul în care lista conține același nume de familie; spre exemplu într-un dicționar, pentru a găsi un cuvant care începe cu litera *f* vom începe căutarea din secțiunea cuvintelor care încep cu aceeași literă. Algoritmii de sortare funcționează de obicei în paralel cu căutarea binară pentru maxima eficiență a căutarii; pentru explicații în detaliu a căutării binare vezi [1], totodată și arborii binari, pentru a înțelege mecanismul din spatele aplicațiilor mari precum Google, Facebook etc. Lista algoritmilor de sortare este vastă, conținând tot felul de algoritmi care deși folosesc diferite mecanici de mutare si ordonare a obiectelor, toți au ca scop o sortare completă standard a elementelor într-un timp cât mai scurt, care nu poate scădea sub complexitatea *O(nLogn)*, decât dacă nu este un algoritm liniar de sortare care funcționează pe un tip local de date, totul fiind explicat în detaliu în [2].

2 Teoria Algoritmilor

În această secțiune voi explica în parte detaliile teoretice ale algoritmilor de sortare, mai precis complexitatea executării (dar nu și complexitatea spațiului de stocare), mecanismul, similitudini cu alți algoritmi și codul propriu-zis, realizat în C++ (pentru o analiză mai detaliată a acestora, vezi [3]), care poate fi accesat de aici sau prin următorul link https://github.com/RaduTM-spec/CompararеALgoritmi. Programul a fost rulat pe Windows 10 (64 bit), calculatorul folosit având un procesor Intel i5-9300h @ 2.40 GHz (folosit în proporție de 14% din puterea maximă asociată frecvenței de bază), și o memorie DDR4 de 8 GB, cu frecvența de 2666 MHz. Rezultatele au fost calculate folosind un cronometru de înaltă precizie din librăria *chrono*.

2.1 Quick Sort

În volumul 5 din [3], paginile 10-16, Hoare descrie *sortarea rapidă* ca o un principiu de a rezolva o problemă prin divizarea ei în două subprobleme mai simple, procesul repetându-se pană când toate problemele rezultate sunt triviale. Aceste probleme triviale sunt apoi rezolvate folosind metode cunoscute, astfel obținând rezultatul problemei originale, care era mult mai complexă. Metoda mai este numită și *Divide and Conquer*, care este des întalnită și în alți algoritmi, precum *Merge Sort*. Procesul *cheie* este partiționarea: fiind dat un tablou de elemente și un pivot x, x este pus pe poziția corectă în tabloul sortat, toate elementele mai mici decât x sunt puse înaintea sa, iar cele mai mari ca el după. Analiza timpului necesar sortării rapide poate fi scrisă:

$$T(n) = T(k) + T(n - k - 1) + \theta(n)$$

Primii doi termeni sunt apeluri recursive, iar ultimul reprezintă procesul de partiționare, unde k este numărul de elemente mai mici decât pivotul. Complexitatea acestui algoritm este *O(nLogn)*, cu un caz nefavorabil de $O(n^2)$. Implementarea implicită este instabilă și folosește spațiu suplimentar în stivă pentru apelurile recursive. În dreapta avem implementarea sortării rapide prin metoda partiționării.

```
int partition (int a[], int beg,
    int end)
{
    int pivot = a[end];
    int i = (beg - 1);

    for (int j = beg; j <= end -
        1; j++)
    {

        if (a[j] < pivot)
        {
            i++;
            swap(&a[i], &a[j]);
        }
    }
    swap(&a[i + 1], &a[end]);
    return (i + 1);
}


void quickSort(int a[], int beg,
    int end)
{
    if (beg < end)
    {

        int pIndex = partition(a,
            beg, end);

        quickSort(a, beg, pIndex
            - 1);
        quickSort(a, pIndex + 1,
            end);
    }
}
```

2.2 Bubble Sort

Începând cu premisa lui Knuth care spune că *"bubble sort pare sa nu aibă nimic ce să-l recomande"*[4], acest algoritm de sortare continuă să fie unul dintre cei mai cunoscuți și folosiți algoritmi, în ciuda complexitații sale exponențiale, $O(n^2)$. Din cauza faptului că sortările necesare în programele uzuale nu conțin un număr mare de elemente (de obicei între 50-150), ineficiența sa nu este remarcată. În schimb, în cazul sortării unui număr mai mare de elemente, bubble sort nu ar putea fi o soluție, timpul executării comparativ cu unul de complexitate *O(nLogn)* ar fi complet diferit. Bubble sort parcurge tabloul de elemente de n ori, apoi de *n-1* ori și tot așa pană la capăt, astfel se ajunge la formula:

$$T(n) = \sum_{i=1}^{n} (n - i)$$

Alături, avem o implementare simplă a sa, dar și una mai eficientă, asemănătoare *Shaker Sort*(eficiența apare într-un singur loc prin prezența unei variabile booleene *swapped* care reține dacă s-a realizat cel puțin o interschimbare pe parcusul celui de-al doilea *for*; în caz contrar algoritmul se oprește, însemnând că toate elementele sunt deja sortate.

```
\\versiunea simpla
void bubbleSort(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]);
}

\\versiunea optimizata
void bubbleSort(int arr[], int n)
{
   int i, j;
   bool swapped;
   for (i = 0; i < n-1; i++)
   {
     swapped = false;
     for (j = 0; j < n-i-1; j++)
     {
        if (arr[j] > arr[j+1])
        {
           swap(&arr[j],
               &arr[j+1]);
           swapped = true;
        }
     }
     if (swapped == false)
        break;
   }
}
```

2.3 Merge Sort

Merge Sort (vezi pagina 49 din [5]) este un reprezentant al sortării în complexitate de *O(nLogn)* și o variantă alternativă, respectiv asemănătoare pentru quick sort; algoritmul folosește aceeași tehnică *Divide and Conquer*, împărțind tabloul în două până se ajunge la un singur element. Majoritatea implementărilor produc o sortare stabilă, adică ordinea elementelor egale se menține din intrare în ieșire. Procesul cheie este *combinarea* și asumarea că sub-tablourile sunt sortate, combinându-le într-un tablou mai mare sortat. Astfel, timpul de executare poate fi reprezentat prin următoarea formulă:

$$T(n) = 2T(n/2) + (n)$$

Codul merge sort este construit mai jos, alăturat având funcția secundară pentru *merging*.

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

```
void merge(int arr[], 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] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    i = 0;
    j = 0;
    k = l;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

2.4 Insertion Sort

Metoda inserării este a patra și ultima sortare testată în această lucrare, care deși nu aduce un punct diferit de analizat față de celelalte două, merită menționat și testat. Algoritmul funcționează după un principiu simplu: fiind un tabloul de sortat parcurs cu index-ul i, în urma lui e se formează un tablou sortat; elementul nou de pe poziția i este pus corespunzator în tabloul din urma sa pentru a se respecta condiția. Complexitatea și timpul de executare sunt asemănătoare *bubble sort*, $O(n^2)$ și respectiv $T(n) = \sum_{i=1}^{n}(n-i)$.

În implementarea de mai jos, odată ce se procesează un nou element, se parcurge înapoi față de poziția i, făcându-se o translatare la dreapta pană când se poate poziționa corect elementul.

```
void insertionSort(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;
    }
}
```

Din punct de vedere teoretic, ne așteptăm la o eficiență substanțială a sortărilor de complexitate *O(nLogn)* (Quick sort si Merge sort) față de cele de $O(n^2)$, cel puțin când vine vorba de numere mari, indiferent de tipurile de date pentru care se face sortarea. Dar, rămâne să observăm concurența dintre algoritmii de aceeași complexitate, care diferă prin anumite metode ce le pot face (nu cu mult) mai rapide.

3 Experiment

Rezultatele testării algorimilor sunt reprezentate prin următoarele tabele, în care sunt prezenți timpii de executare calculați în milisecunde, respectiv secunde, în funcție de numarul de elemente (care variază de la 10 la 100.000.000). Pentru fiecare numar de elemente au fost realizate minim 3 teste, în unele situații timpul de executare în milisecunde fiind constant (din cauza deficienței acurateței maximale - până la nanosecunde), precizat o singură dată, sau timpul de executare fiind neatins deoarece nu ar fi fost realizat în timp util (în cazul aloritmilor de complexitate $O(n^2)$ sortarea unui numar prea mare este imposibilă de realizat în practică). Au fost realizate teste pe un set *random* de date și din mulțimea {0,1}.

3.1 Quick Sort

Sortare de 10 - 100.000.000

quicksort	1.000.000 elemente Element maxim: 1M	10.000.000 elemente Element maxim: 1M	100.000.000 elemente Element maxim: 1M
Timp executare in Milisecunde	130.108 ms 126.905 ms 126.653 ms	3351.55 ms 3321.02 ms 3325.99 ms	38285.71 ms
Timp executare in Secunde	0.130108 s 0.126905 s 0.126653 s	3.35155 s 3.32102 s 3.32599 s	38.28571 s

10 elemente Element maxim:10	100 elemente Element maxim:100	1000 elemente Element maxim :1000
0.0006 ms	0.0049 ms 0.0051 ms 0.0047 ms	0.064 ms 0.0651 ms 0.0752 ms
6e-07 s	4.9e-06 s 5.1e-06 s 4.7e-06 s	6.4e-05 s 6.51e-05 s 7.52e-05 s

Sortare pe {0,1}

Quick_sort	1000 elemente Elemente din {0,1}	1000000 elemente Elemente din {0,1}	10.000.000 elemente Elemente din {0,1}
Timp executare in Milisecunde	0.5179 ms 0.5147 ms 0.6992 ms	1060.52 ms 1056.02 ms 1081.18 ms	*- timpi prea mari de executie (aprox 18 000ms)*
Timp executare in Secunde	0.0005179 s 0.0005147 s 0.0006992 s	1.06052 s 1.05602 s 1.08118 s	*- timpi prea mari de executie (aprox 18 minute)*

3.2 Bubble Sort

Sortare de 10 - 100.000.000

bubblesort	1.000.000 elemente Element maxim:1M	10.000.000 elemente Element maxim: 1M	100.000.000 elemente Element maxim: 1M
Timp executare in Milisecunde	*7800000 ms*	*-timpi prea mari de executie*	*-timpi prea mari de executie*
Timp executare in Secunde	*7800 s* *Aprox. 130 minute*	*-timpi prea mari de executie*	*-timpi prea mari de executie*

10 elemente Element maxim:10	100 elemente Element maxim:100	1000 elemente Element maxim:1000
0.002 ms 0.0018 ms 0.0019 ms	0.1108 ms 0.0724 ms 0.0828 ms	7.8543 ms 7.839 ms 7.7413 ms
2e-06 s 1.8e-06 s 1.9e-06 s	0.0001108 s 7.24e-05 s 8.28e-05 s	0.0078543 s 0.007839 s 0.0077413 s

Sortare pe {0,1}

Bubble_sort	1000 elemente Elemente din {0,1}	1000000 elemente Elemente din {0,1}	10.000.000 elemente Elemente din {0,1}
Timp executare in Milisecunde	4.5085 ms 4.7984 ms 4.7822 ms	*- timpi prea mari de executie*	*- timpi prea mari de executie*
Timp executare in Secunde	0.0045085 s 0.0047984 s 0.0047822 s	*- timpi prea mari de executie*	*- timpi prea mari de executie*

3.3 Merge Sort

Sortare de 10 - 100.000.000

mergesort	1.000.000 elemente Element maxim: 1M	10.000.000 elemente Element maxim: 1M	100.000.000 elemente Element maxim: 1M
Timp executare in Milisecunde	508.363 ms 514.552 ms 494.197 ms	5630.9 ms 5694.79 ms 5219.85 ms	51428.3 ms
Timp executare in Secunde	0.508363 s 0.514552 s 0.494197 s	5.6309 s 5.69479 s 5.21985 s	51.4283 s

10 elemente Element maxim:10	100 elemente Element maxim:100	1000 elemente Element maxim :1000
0.0076 ms 0.0092 ms 0.0083 ms	0.0662 ms 0.086 ms 0.0678 ms	0.5757 ms 0.8018 ms 0.5545 ms
7.6e-06 s 9.2e-06 s 8.3e-06 s	6.62e-05 s 8.6e-05 s 6.78e-05 s	0.0005757 s 0.0008018 s 0.0005545 s

Sortare pe {0,1}

Merge_sort	1000 elemente Elemente din {0,1}	1000000 elemente Elemente din {0,1}	10.000.000 elemente Elemente din {0,1}
Timp executare in Milisecunde	0.4939 ms 0.6176 ms 0.5326 ms	432.658 ms 459.91 ms 434.934 ms	4680.94 ms 4541.77 ms 4649.23 ms
Timp executare in Secunde	0.0004939 s 0.0006176 s 0.0005326 s	0.432658 s 0.45991 s 0.434934 s	4.68094 s 4.54177 s 4.64923 s

3.4 Insertion Sort

Sortare de 10 - 100.000.000

insertionSort	1.000.000 elemente Element maxim: 1M	10.000.000 elemente Element maxim: 1M	100.000.000 elemente Element maxim: 1M
Timp executare in Milisecunde	500.000 ms	-timpi prea mari de executie	-timpi prea mari de executie
Timp executare in Secunde	500 s Aprox. 8.33 minute	-timpi prea mari de executie	-timpi prea mari de executie

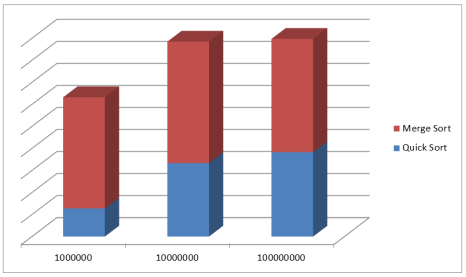
10 elemente Element maxim:10	100 elemente Element maxim:100	1000 elemente Element maxim:1000
0.0003 ms 0.0002 ms 0.0013 ms	0.0065 ms 0.0086 ms 0.006 ms	0.4935 ms 0.5042 ms 0.4998 ms
3e-07 s 2e-07 s 1.3e-06 s	6.5e-06 s 8.6e-06 s 6e-06 s	0.0004935 s 0.0005042 s 0.0004998 s

Sortare pe {0,1}

Insertion_sort	1000 elemente Elemente din {0,1}	1000000 elemente Elemente din {0,1}	10.000.000 elemente Elemente din {0,1}
Timp executare in Milisecunde	0.264 ms 0.2546 ms 0.254 ms	256345 ms	*- timpi prea mari de executie*
Timp executare in Secunde	0.000264 s 0.0002546 s 0.000254 s	256.345 s	*- timpi prea mari de executie*

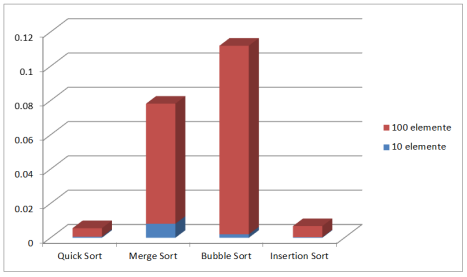
4 Discuție

În urma examinării rezultatelor, în cazul mulțimii de date *random* din setul larg de date, se pot observa următoarele aspecte: Quick Sort și Merge Sort sunt mult mai rapide decât celelalte două, din cauza complexității. Comparând mai întai rezultatele dintre acești doi algoritmi se poate observa că deși Quick Sort are o eficiență sporită în cazul unui număr mic de elemente, diferența începe să se echilibreze pe măsură ce numărul de elemente crește. Pentru vizualizarea acestei echilibrări am realizat acest tabel care indică proporțiile de timp executat dintre cei doi algoritmi, pentru cele mai largi tablouri sortate.



Contrar, în cazul Bubble Sort și Insertion Sort, pe măsura creșterii

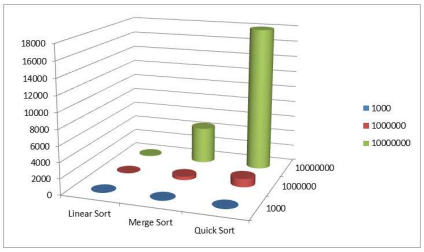
numărului de elemente diferența dintre cei doi crește, astfel Insertion Sort fiind mai eficient. Diferența dintre cele două tipuri de algoritmi (în funcție de complexitate) este vizibilă, odată ajunși la 10 milioane de elemente tipul de executare pentru un algoritm de complexitate $O(n^2)$ fiind irealizabil în timp util. Totuși, în cazul a 10 sau 100 de elemente diferența dintre Quick Sort și Insertion Sort spre exemplu, nu este considerabilă, conform predicțiilor teoretice exemplificate în secțiunea precedentă. Mai jos, avem o tabelă cu statisticile celor 4 algoritmi în cazul unor sortări de 10, respectiv 100 de elemente.



Pentru sortarea unor tablouri ce conțin doar elemente din mulțimea { 0,1}, am realizat mai jos un tablou care sortează în timp liniar, pentru a observa distanțarea timpului de execuție a algoritmilor față de timpul de execuție în complexitate $O(n)$ pe măsură ce mărimea tabloului crește:

Linear_sort	1000 elemente Elemente din {0,1}	1000000 elemente Elemente din {0,1}	10.000.000 elemente Elemente din {0,1}
Timp executare in Milisecunde	0.0078 ms 0.0081 ms 0.0079 ms	7.7099 ms 7.7112 ms 7.6857 ms	77.8286 ms 77.2049 ms 81.5903 ms
Timp executare in Secunde	7.8e-06 s 8.1e-06 s 7.9e-06 s	0.0077099 s 0.0077112 s 0.0076857 s	0.0778286 s 0.0772049 s 0.0815903 s

Diferența dintre un algoritm ce sortează în timp liniar și ceilalți algoritmi este vizibilă în numere, dar am realizat și un grafic pentru a vizualiza acest lucru. După cum se observă, Merge Sort pare a fi mai eficient decât Quick Sort pe măsură ce datele de intrare sunt de un număr mai mare; coloanele reprezentative pentru rezultatele din sortarea liniară par discuri comparativ cu ale celorlalți algoritmi. Totuși, asta nu înseamnă că algoritmii liniari sunt neapărat eficienți, deoarece nu s-a discutat despre complexitatea spațiului folosit.



5 Concluzii

în final, am ajuns la concluzia că algoritmii de sortare din complexitatea *O(nLogn)* sunt mai eficienți chiar și in cazul unui numar mic de date. Algoritmii cu $O(n^2)$ ar putea fi folosiți totuși pe un număr mai mic decât 100 dacă se urmărește o implementare rapidă și ușoară, de preferat Insertion sort în locul Bubble sort. În situația unor date de intrare mari, Quick sort sau Merge sort (sau alți algoritmi de aceeași complexitate) sunt o alegere nu doar necesară, ci obligatorie. Implementarea fără partiționare este mai bună și mai eficientă. Quick sort rămâne în general cel mai bun algoritm de sortare, meritând să fie folosit indiferent de dimensiunea tabloului care trebuie sortat sau de tipul de date pe care îl conține.

6 Studiu de viitor

1. Adăugarea mai multor algoritmi de sortare precum Heap Sort, Radix Sort, Counting Sort, în scopul comparării cu celelalte metode folosind aceleași date de intrare.
2. Comparare mai detaliată a algoritmilor din mai multe perspective, evidențiind toate proprietățile necesare: complexitatea spațiului de stocare, cea mai bună și cea mai proastă performanță, stabilizare etc.
3. Testare pe mai multe seturi de date de natură diferită: șiruri de caractere, dicționare, cuvinte sau numere foarte mari.
4. Realizarea mai multor tabele si grafice mai sugestive și mai precise, atât pentru noile testări cât și pentru îmbunătățirea lucrării.

Bibliografie

[1] T. Cormen, T.H., Leiserson, C.E., Rivest, R.L. *Introduction to Algorithms (2^{nd} ed.)* (1 Septembrie 2001).

[2] P. Mridha, B.J. Datta. *Algorithm for Analysis the Time Complexity for Iterated Local Search* (28 Iunie 2021)

[3] C.A.R. Hoare, *The Computer Journal* (1 Ianuarie 1962)

[4] D.E. Knuth, The dangers of computer science theory, *Logic, Methodology and Philosophy of Science 4* (1973)

[5] D. Grover and S. Beniwal, *Performance Analysis of Merge Sort and Quick Sort: MQSORT*, IJCSMS International Journal of Computer Science Management Studies, Vol. 13 (Iulie 2013)