Sortarea algoritmilor de sortare prin experiment și comparație teoretică

Valentin Cosmin Cîrju
Departamentul de Informatică,
Facultatea de Matematică şi Informatică,
Universitatea de Vest Timişoara, România,
Email: valentin.cirju03@e-uvt.ro

Mai 2023

Rezumat

Această lucrare are ca obiectiv compararea eficienței diferiților algoritmi de sortare, prin analiza atât a teoriei, cât și a performanței lor în practică. Sortarea este o sarcină fundamentală în domeniul informaticii, iar eficiența devine o problemă majoră atunci când trebuie să sortăm cantități mari de date. În această lucrare, se vor analiza proprietățile implementărilor reale ale algoritmilor de sortare și se vor administra experimente ample pentru a obține date empirice despre eficiența lor practică în diferite situații. Scopul este de a oferi o comparare detaliată din perspectiva ordinii de complexitate, timpului de execuție și numărului de interschimbări a diferiților algoritmi de sortare, cu accent pe performanța lor practică, și de a încheia cu constatări teoretice. Lucrarea va fi utilă în dezvoltarea și optimizarea aplicațiilor informatice care implică sortarea unor cantități mari de date.

Cuprins

Listă de tabele

1 Introducere

Sortarea unui tablou reprezintă procesul de organizare a unui șir de elemente într-o anumită ordine, astfel încât acestea să fie mai ușor de utilizat sau de găsit într-un anumit context. În viața reală, acest proces este adesea utilizat pentru a căuta elemente cu anumite caracteristici comune, într-un set mare de date. Sortarea poate fi efectuată prin aplicarea unui set de reguli sau criterii de ordonare a elementelor, în funcție de nevoile și preferințele utilizatorului. În cazul numerelor, de exemplu, o metodă comună de sortare este cea în ordine crescătoare sau descrescătoare, în funcție de nevoile specifice ale utilizatorului.

1.1 Scopul cercetării

Scopul principal al acestei cercetări este de a realiza o analiză exhaustivă a algoritmilor de sortare prin intermediul unui studiu al literaturii și a implementărilor potențiale. În cadrul cercetării, algoritmii vor fi definiti și explicați în detaliu, iar eficiența și celelalte proprietăți ale acestora vor fi analizate și comparate într-un mod riguros. Prin consultarea literaturii specializate, vor fi investigatate diverse variante și îmbunătățiri ale algoritmilor fundamentali, iar avantajele și dezavantajele asociate acestora vor fi sintetizate și dezbătute. De asemenea, această lucrare își propune să colecteze date referitoare la timpul de execuție al diferitilor algoritmi de sortare, în vederea comparării practicabilității variantelor lor de implementare. Cantitățile relevante includ numărul de operații de diferite tipuri efectuate, timpul de rulare absolut și consumul de spațiu de memorie.

Scopul ultim este să se acumuleze și să se evalueze date care să permită realizarea unei analize detaliate a eficacității practice a algoritmilor și a implementărilor lor concrete. Experimentul va fi replicat pe mai multe cazuri de testare, incluzând diferite tipuri de date de intrare, cum ar fi matrice de numere și șiruri de dimensiuni variabile.

1.2 Exemple în practică

Iată câteva exemple care ilustrează avantajele sortării și a căutării într-o listă ordonată în comparație cu o listă neordonată:

Atunci când este necesar să eliminăm duplicatele dintr-o listă de un milion de elemente, este mult mai eficient să sortăm aceste elemente înainte de a începe procesul de eliminare a duplicatelor. Această abordare reduce semnificativ timpul necesar pentru eliminarea duplicatelor, în comparație cu procesarea unei liste neordonate, în care fiecare element trebuie comparat cu toate celelalte elemente pentru a identifica duplicatul.

Un alt exemplu important este acela al căutării de date într-o listă. În cazul în care lista nu este sortată conform unui criteriu specific, căutarea unei date necesită o analiză exhaustivă a întregii liste. În schimb, dacă lista este deja sortată, putem localiza rapid elementul dorit într-o anumită secțiune a listei.

Un exemplu semnificativ este reprezentat de platformele de socializare, unde cele mai relevante postări sunt afișate pe pagina principală pentru utilizatori. Acest proces de afișare a postărilor relevante este realizat prin sortarea tuturor postărilor disponibile în funcție de gradul lor de relevanță și afișarea acestora în ordine descrescătoare a relevanței. Acest principiu este de asemenea aplicat și în procesul de căutare a cuvintelor cheie în cadrul platformei de socializare.

În ceea ce privește motorul de căutare Google, se poate constata că timpul mediu de căutare pentru 100.000.000 de rezultate este mai mic de un minut, în timp ce pentru Facebook, care are un număr de aproximativ 24 de miliarde de rezultate, se așteaptă un timp de căutare de aproximativ 0,62 secunde.

2 Prezentare formală a problemei și soluției

De la timpurile cele mai vechi, organizarea bunurilor prin sortare sau grupare pe categorii a reprezentat o formă simplă de organizare sistematică. Această problematică a fost de interes pentru cercetători încă de la apariția primului calculator digital programabil (ENIAC, 1945). Sortarea se referă la procesul de ordonare a unei liste de elemente în funcție de un criteriu specific, iar eficiența sortării în ceea ce privește memoria ocupată și timpul de execuție este crucială pentru interpretarea datelor într-un mod cât mai facil.

2.1 Preliminarii

În acest studiu, vom examina șapte algoritmi de sortare: Insertion Sort, Selection Sort, Bubble Sort, Merge Sort, Radix Sort, Heap Sort și Quick Sort. Acești algoritmi sunt larg utilizați în sortarea listelor de elemente și prezintă diferențe semnificative în privința eficienței și a performanței în diverse scenarii de utilizare.

- Bubble Sort, cunoscut și sub denumirea de "sortarea prin interschimbarea elementelor vecine", a primit acest nume din partea lui Kenneth E. Iverson [Iverson(1962)]. Este considerată cea mai simplă metodă de sortare, care poate fi implementată cu ușurință. Cu ajutorul a două structuri se parcurge tabloul și se compară pe rând fiecare element A cu restul elementelor din listă. În cazul sortării elementelor în ordine crescătoare, interschimbarea se va face doar dacă elementul A comparat cu restul elementelor este mai mic decât elementul curent.

- Conform spuselor lui Donald Knuth în cadrul cursului "The Theory and Techniques for Design of Digital Computers" susținut de John Mauchly [Eckert(1946)], Insertion Sort este un algoritm simplu, similar cu cel de Selection Sort. Acesta constă în compararea, rând pe rând, a elementelor tabloului, începând cu al doilea "element curent". Elementele din stânga elementului curent sunt comparate cu acesta, de la dreapta la stânga (începând cu elementul din stânga celui curent și terminând cu primul element), iar elementele mai mari decât elementul curent vor fi mutate cu o poziție spre dreapta.

- Selection Sort este un algoritm de sortare, descris de Donald Knuth în cartea sa "*The Art of Computer Programming*" [Knuth(1998)]. Pentru a implementa acest algoritm, se utilizează două variabile, una pentru elementul curent și alta pentru elementul minim. Inițial, aceste variabile vor fi setate la primul element din tablou. Se parcurge apoi tabloul, fiecare element fiind luat pe rând ca fiind "elementul curent". Dacă elementul curent este mai mic decât minimul, variabila minim va fi actualizată cu valoarea elementului curent, iar căutarea va continua în tablou. La final, elementul inițial pentru care am făcut căutarea va fi interschimbat cu minimul și se va continua căutarea elementului minim față de elementul de pe următoarea poziție din tablou.

- Radix sort, un algoritm de sortare cu o istorie îndelungată, a fost menționat pentru prima dată în lucrarea lui Herman Hollerith intitulată "Mașinile de tabelare și metodele lor" [Hollerith(1890)] publicată în 1890. Această lucrare a reprezentat un moment de pionierat în dezvoltarea mașinilor de tabelare și a introdus conceptul de sortare radix. Ulterior, algoritmii de sortare radix au fost utilizați pe scară largă pentru sortarea cardurilor perforate încă din 1923, ceea ce a avut un impact semnificativ asupra procesării datelor și a tehnicilor de calcul. În prezent, sortarea Radix continuă să fie folosită în domenii precum sortarea șirurilor binare și a numerelor întregi, demonstrând eficiență și relevanță în era modernă a prelucrării datelor.

- Sortarea prin Interclasare (Merge Sort), inventată de John von Neumann în 1945 [Knuth(1998)], este o metodă recursivă care se bazează pe principiul "divide et impera". Tabloul este împărțit în două subtablouri, acest proces repetându-se cu subtablourile obținute până se ajunge la subtablouri ce conțin un număr minim de elemente. Ulterior, subtablourile sunt sortate pe rând, două câte două, iar din acestea se obțin subtablouri sortate.

- Heap sort a fost introdus de către Robert W. Floyd și J.W.J. Williams în anul 1964 [Robert W. Floyd(1964)]. Heapsort este clasificat ca un algoritm de sortare "in place", ceea ce înseamnă că modificările necesare pentru a sorta o listă dată sunt efectuate direct în spațiul de memorie original, fără a necesita spațiu suplimentar, acest fapt îl face potrivit în special pentru situațiile în care utilizarea suplimentară de memorie este limitată sau nu este fezabilă. Algoritmul folosește o structură de date numită heap (sau arbore binar), care reprezintă un arbore binar complet în care fiecare nod respectă proprietatea de heap. Prin utilizarea structurii de date heap, Heap sort își atinge scopul de sortare prin construirea unui heap din lista de intrare și extragerea repetată a elementului maxim din heap și plasarea acestuia la sfârșitul porțiunii sortate a listei. Elementele extrase sunt apoi interschimbate pentru a menține proprietatea de heap, asigurând că elementele rămase sunt în ordinea corectă. Acest proces continuă până când toate elementele sunt sortate în ordine crescătoare. Deși Heap sort nu este cel mai eficient algoritm de sortare în ceea ce privește timpul de execuție, acesta prezintă o complexitate medie de timp de $O(n \log n)$.

- Quick Sort este un algoritm de sortare dezvoltat de Tony Hoare în 1961 [Hoare(1961)], care utilizează un pivot pentru a separa elementele dintr-un tablou în două sub-tablouri: unul cu elemente mai mici decât pivotul și unul cu elemente mai mari decât pivotul. Alegerea pivotului poate fi realizată prin selectarea unui element din tablou sau prin utilizarea primului sau ultimului element din tablou.

  Procedura constă în găsirea a două elemente: un element din stânga care este mai mare decât pivotul și un element din dreapta care este mai mic decât pivotul. Atât timp cât cele două elemente sunt găsite și indexul elementului din stânga este mai mic decât indexul elementului din dreapta, cele două elemente sunt interschimbate. În caz contrar, pivotul este interschimbat cu "elementul din stânga" și tabloul este împărțit în două sub-tablouri: unul de la primul element până la pivot (exclusiv) și celălalt de la elementul din dreapta pivotului până la ultimul element al tabloului. Procedura este apoi repetată pentru fiecare sub-tablou, până când întregul tablou este sortat.

3 Modelare și implementare

Problema propusă urmează să fie investigată printr-un experiment în care se va utiliza limbajul de programare C pe platforma open-source CLion. Obiectivul principal constă în analizarea și comparația comportamentului algoritmilor de sortare în ceea ce privește stocarea memoriei. Este important de menționat că experimentul se va desfășura fără a modifica cantitatea de memorie disponibilă, astfel încât să se poată evidenția mai precis diferențele între algoritmii de sortare în acest aspect specific al memoriei.

În scopul evaluării comportamentului algoritmilor de sortare, am creat serii de ansambluri de tablouri compuse din 10, 100, 1.000, 10.000 și 100.000 de tablouri individuale pentru fiecare set în parte. Fiecare tablou conține 10, 100, 1.000 sau 10.000 de elemente, respectiv, în funcție de setul corespunzător.

Pentru fiecare serie de ansambluri, am aplicat următoarele configurații de sortare:

- primul tablou este sortat în ordine crescătoare, începând de la 1 și continuând până la 10, de la 1 la 100, și așa mai departe, în funcție de numărul de elemente din tablou;

- al doilea tablou este sortat parțial crescător (unde la aproximativ 20% din tablou este inserat un număr aleatoriu între 0 și 100);
- al treilea tablou este sortat în ordine descrescătoare, începând de la 10 și continuând până la 1, de la 100 la 1, și așa mai departe, în funcție de numărul de elemente din tablou;
- al patrulea tablou constă într-un "tablou plat" (format dintr-un număr mic de elemente care se repetă, în acest caz, numerele de la 1 la 3).
- celelalte tablouri au fost generate aleatoriu, conținând numere cuprinse între 0 și 100.

Notă: Secvențele de cod pot fi vizualizate pe următorul link - Testing-of-sorting-Algorithms.git. Listele folosite pentru sortari: Google Drive - Lists

4 Experiment

Rezultatele experimentale pot fi observate în tabela 1 și 2. Datele de testare sunt prezentate în secunde și sunt conforme cu așteptările. Pentru un număr mic de elemente (10-1000), rezultatele sunt similare, cu diferențe între intervale destul de mici. În cazul unui număr mare de elemente (1000-10000), diferențele în timpul necesar pentru executarea programelor sunt semnificativ mai mari. Pentru un set de date care conține 10000 de tablouri a 10000 de elemente sortate aleatoriu, așa cum se poate observa în tabela 1, algoritmul Bubble Sort necesită aproximativ 2400 de secunde (40 minute) pentru a rula, în contrast cu aproximativ 19 secunde necesare algoritmilor Merge Sort, Heap Sort și QuickSort. Din păcate, memoria necesară pentru a rula cu succes algoritmul Merge Sort pentru un set de date de 1000000 de elemente sortate în fiecare caz, precum și algoritmul QuickSort pentru seturi de date mai mari de 100000 de elemente sortate în ordine crescătoare sau descrescătoare, sau mai mari de 100000 de elemente pentru seturi de date plate, și setul de date de 1000000 de elemente sortate aleatoriu depășește memoria disponibilă pe platforma utilizată inițial. Problema memoriei insuficiente este destul de serioasă, fiind într-o oarecare măsură mai importantă decât problema timpului de executare, deoarece programul nu se va finaliza. Aceasta poate fi remediată prin adăugarea de memorie.

4.1 Timp de execuție

Timpul de execuție este măsurat și cuantificat în cadrul acestei cercetări prin intermediul conceptului de Ceas de Perete (Wall Clock Time), o metrică care reflectă durata totală a timpului petrecut în execuția unui program, inclusiv timpul de așteptare și de întreruperi. Această abordare particulară în determinarea duratei a fost adoptată ca rezultat al unei preferințe personale și a unei curiozități care își propune să exploreze aspecte mai largi ale performanței și eficienței algoritmilor de sortare.

Mai exact, timpul măsurat reprezintă intervalul necesar pentru a sorta toate seturile de date utilizate în cadrul experimentului, excluzând în mod deliberat procesul de încărcare a fișierelor și verificarea corespunzătoare a seturilor de date sortate. Această alegere metodologică are scopul de a izola și evalua în mod specific performanța procesului de sortare propriu-zis, eliminând eventuale influențe externe asociate cu etapele de pregătire și validare a datelor.

Tabela 1: Timp de execuție pentru fiecare algoritm de sortare în parte.

Seturi \El. tablou	Bubble	Insertion	Selection	Radix	Merge	Heap	Quick
10x10	0.000004	0.000002	0.000003	0.000005	0.000015	0.000005	0.000003
10x100	0.000209	0.000051	0.000122	0.000048	0.000160	0.000084	0.000084
10x1000	0.017863	0.004547	0.009745	0.000454	0.001600	0.001187	0.005147
10x10000	1.996335	0.443294	0.945542	0.004986	0.016491	0.015183	0.467771
100x10	0.000057	0.000016	0.000030	0.000051	0.000112	0.000037	0.000035
100x100	0.002474	0.000619	0.001242	0.000448	0.001408	0.000874	0.000542
100x1000	0.179254	0.050669	0.098128	0.004930	0.015545	0.012694	0.011641
100x10000	23.554276	5.000114	9.470841	0.048128	0.179201	0.165071	0.608462
1000x10	0.000325	0.000150	0.000272	0.000515	0.000939	0.000338	0.000325
1000x100	0.024558	0.006220	0.012826	0.004536	0.013279	0.008058	0.005184
1000x1000	1.818231	0.515861	0.981253	0.047768	0.160315	0.127031	0.076320
1000x10000	239.626285	50.533495	94.796886	0.479238	1.799572	1.682816	2.042202
10000x10	0.003008	0.001632	0.002674	0.005267	0.008823	0.003349	0.002724
10000x100	0.246479	0.061927	0.128122	0.044781	0.131066	0.080123	0.051788
10000x1000	18.085748	5.157861	9.809161	0.480112	1.600048	1.272713	0.724196
10000x10000	2413.675219	503.655559	941.130039	4.809519	22.476264	16.770294	16.256385
100000x10	0.032809	0.015892	0.027032	0.052912	0.094061	0.034275	0.027563
100000x100	2.491388	0.614728	1.271063	0.452129	1.285657	0.796751	0.512606
100000x1000	179.424047	51.160142	97.488193	4.789265	18.035928	12.821038	7.142425

4.2 Numărul de interschimburi

Numărul de interschimburi este măsurat într-un mod diferit pentru fiecare algoritm, iar inputul furnizat face o mare diferență în rezultate. Din acest motiv, următoarea tabelă va prezenta o medie a 10 teste, cu 10 inputuri diferite.
Notă: OF = OverFlow (sau capacitate a memoriei depășită)

Tabela 2: Număr de interschimbări pentru fiecare algoritm de sortare.

Seturi \El. tablou	Bubble	Insertion	Selection	Radix	Merge	Heap	Quick
10x10	212	212	60	170	340	275	224
10x100	21166	21166	775	2400	6720	5727	10237
10x1000	2165285	2165285	8163	28000	99760	88897	785955
10x10000	216730415	216730415	81839	310000	1336160	1213506	75360637
100x10	2379	2379	709	2100	3400	2710	1804
100x100	243692	243692	9219	26600	67200	57952	42531
100x1000	24429875	24429875	96781	298000	997600	901516	1227490
100x10000	OF	OF	972332	3010000	13361600	12310402	79978177
1000x10	22335	22335	6975	20940	34000	27550	18277
1000x100	2446722	2446722	94033	263900	672000	579852	374821
1000x1000	246861552	246861552	983063	2998000	9976000	9028206	5682682
1000x10000	OF	OF	9879217	30010000	133616000	123292126	126548769
10000x10	222315	222315	70084	209960	340000	275546	183834
10000x100	24501292	24501292	942398	2616700	6720000	5800490	3686784
10000x1000	OF	OF	9846920	29998000	99760000	90300602	50111619
10000x10000	OF	OF	98939396	300010000	1336160000	1233075049	592057557
100000x10	2226978	2226978	702619	2093970	3400000	2755392	1833124
100000x100	245064551	245064551	9429161	26302300	67200000	57995362	36812717
100000x1000	OF	OF	98490047	299988000	997600000	902981076	494192809

5 Comparație cu literatura

Această lucrare a fost puternic inspirată de următoarele cărți, care mi-au conferit o încredere solidă în corectitudinea soluției mele:

• "Introduction to Algorithms" de Thomas H. [Cormen(1990)], Charles E. Leiserson, Ronald L. Rivest și Clifford Stein: Acest manual clasic este considerat pe scară largă resursa definitivă în domeniul algoritmilor. Acesta acoperă în mod exhaustiv algoritmii de sortare, inclusiv sortarea prin inserție, sortarea prin selecție, sortarea prin interclasare, quicksort și altele. Cartea oferă explicații detaliate, pseudocod și analiză a algoritmilor.

• "The Algorithm Design Manual" de [Skiena(1997)]: Această carte oferă o prezentare cuprinzătoare a diferitelor algoritmi, inclusiv algoritmii de sortare. Aceasta acoperă tehnici fundamentale pentru proiectarea și analizarea algoritmilor, oferind perspective asupra momentului în care să folosiți algoritmi de sortare specifici și cum să-i optimizați. Cartea include, de asemenea, exemple practice și exerciții.

• "Sorting and Searching" de [Knuth(1998)]: Facând parte din seria "Arta Programării", această carte se aprofundează în algoritmii de sortare și căutare. Explorează diferite tehnici de sortare și analiza acestora, inclusiv sortarea prin inserție, sortarea prin metoda bulelor, quicksort și îmbinarea multiplă. Stilul de scriere al lui Knuth este extrem de detaliat și riguros, ceea ce o face potrivită pentru cititorii interesați de o înțelegere exhaustivă a subiectului.

Pentru a înțelege mai bine rezultatele experimentului, avem nevoie de rezultatele teoretice existente. Dacă am face o comparație a numărului de interschimbări efectuate în momentul rulării programului, acestea ar fi corecte din punct de vedere teoretic.

Tabela 3: Ordin de complexitate al fiecărui algoritm.

	Bubble	Insertion	Selection	Radix	Merge	Heap	Quick
Caz favorabil	$\Omega(n)$	$\Omega(n)$	$\Omega(n)$	$\Omega(nk)$	$\Omega(n \log n)$	$\Omega(n \log n)$	$\Omega(n \log n)$
Caz mediu	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(nk)$	$\Theta(n \log n)$	$\Theta(n \log n)$	$\Theta(n \log n)$
Caz defavorabil	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(nk)$	$O(n \log n)$	$O(n \log n)$	$O(n^2)$

6 Concluzii și direcții viitoare

Tema centrală a acestei discuții se referă la problema sortării elementelor, care derivă din necesitatea de a găsi și ordona elementele cu caractere asemănătoare. Sortarea este considerată soluția optimă pentru abordarea eficientă a acestei probleme. Procesul de sortare implică aranjarea elementelor care sunt inițial distribuite în mod aleatoriu într-un mod ordonat, fie în ordine crescătoare sau descrescătoare, fie de la cel mai semnificativ la cel mai nesemnificativ și invers, utilizând un criteriu specific.

În această lucrare, am implementat algoritmii și am analizat timpul necesar pentru executarea acestora, luând în considerare două criterii: modul inițial de sortare a elementelor și numărul de elemente ce trebuie sortate (am examinat 19 seturi de date, cu dimensiuni variind de la 10 la 1.000.000 de elemente).

Rezultatele obținute au fost în mare măsură satisfăcătoare, cu toate că am observat importanța unei memorii generoase, dată fiind lipsa de rezultate concludente pentru algoritmii Merge Sort, Heap Sort și QuickSort.

Conform așteptărilor, algoritmii Merge Sort, Heap Sort și QuickSort au prezentat performanțe mai bune decât algoritmii Insertion Sort, Selection Sort și Bubble Sort datorită timpului de execuție de complexitate $O(n \log n)$. Avantajul deosebit al acestor trei algoritmi este evidențiat prin eficiența lor temporală, însă în cazul seturilor de date cu un număr mare de elemente, problema memoriei insuficiente poate deveni o provocare. Pentru a depăși această problemă, putem considera adăugarea de memorie suplimentară sau chiar îmbunătățirea algoritmilor existenți, dacă există oportunitatea de a face acest lucru.

Bibliografie

[Iverson(1962)] Kenneth E. Iverson. A programming language. In *Proceedings of the May 1-3*. AFIPS, 1962.

[Eckert(1946)] P. Eckert. The theory and techniques for design of digital computers. In *The theory and techniques for design of digital computers*. Springer, 1946.

[Knuth(1998)] Donald E. Knuth. The art of computer programming. In *Volume 3: Sorting and Searching" (2nd edition, 1998)*. Addison-Wesley, 1998.

[Hollerith(1890)] Herman Hollerith. Tabulating machines and their methods. In *The Massachusetts Institute of Technology*. MIT, 1890.

[Robert W. Floyd(1964)] J.W.J. Williams Robert W. Floyd. Heap sort: A fast sorting algorithm in place. In *Association for Computing Machinery*. ACM, 1964.

[Hoare(1961)] Charles Antony Richard Hoare. Algorithm 64: Quicksort. In *Communications of the ACM, 4(7):321*. Association for Computing Machinery, 1961.

[Cormen(1990)] Thomas H. Cormen. Introduction to algorithms, 3rd edition. In *Introduction to Algorithms*. MIT Press, 1990.

[Skiena(1997)] Skiena. The algorithm design manual. In *The Algorithm Design Manual*. Springer, 1997.