Comparație experimentală a algoritmilor de sortare

-Claudiu-Andras Kocsis-
Faclutatea de Matematică și Informatică
Universitatea de Vest din Timișoara
Email: claudiu.kocsis02@e-uvt.ro

May 2022

Abstract

Scopul principal al acestei lucrări este de a compara timpii de rulare ai algoritmilor de sortare predați și utilizați în domeniul informaticii. Algoritmii utilizați în această lucrare sunt: Bubble Sort, Selection Sort, Insertion Sort, Quick Sort, Merge Sort, Heap Sort, Counting Sort și Radix Sort. Acest experiment folosește o serie de numere întregi pozitive cuprinse între 0 și 1 milion, cu dimensiuni cuprinse între 1000 și 1 milion, iar codul sursă este scris în C++.
Lucrarea compară mai întâi algoritmii dintr-o persectiva teoretică în funcție de complexitatea O, în timp ce compararea practică și experimentală ocupă restul lucrării.
Rezultatele erau, în general, previzibile, dar au existat si câteva surprize: Merge Sort s-a dovedit a fi mai lent decât Bubble, Selection și Insertion pentru liste de dimensiuni mici, și a fost, per ansamblu, cel mai lent algoritm comparativ cu Counting si Radix Sort care au fost cei mai rapizi pentru liste mari de numere, dar nu și liste mai mici, fiind depășiți de algoritmi mai lenți.
Cât despre alte concluzii, Quick Sort and Heap Sort au fost rapizi și stabili; Bubble Sort a devenit tot mai ineficient pe măsură ce lista de numere a devenit tot mai mare, Selection and Insertion Sort au devenit, de asemenea, mai ineficienți dar după o durata mai îndelungată de timp; Merge Sort pare destul de rapid pentru liste de mărimi medii, dar este destul de ineficient în general, iar Counting si Radix Sort sunt cele mai rapide pentru liste de dimensiuni mari.

Cuprins

1 Introducere

1.1 Motivația

Obiectivul principal al acestei lucrări este să compare timpii de rulare ai algoritmilor folosiți în domeniul algoritmicii. O parte esențială al acestui domeniu este de a găsi cele mai eficiente soluții pentru problemele întâlnite în acesta; sortarea datelor în structuri de date precum liste de numere este una dintre cele mai comune provocări cu care se confruntă oamenii din acest domeniu.

Sortarea structurilor de date este o problemă întâlnită în toate colțurile acestui domeniu: sortarea jucătorilor după nume/ scor într-un joc video, sortarea persoanelor după nume/ carte de identitate/ istoric medical într-o bază de date medicală sau sortarea unei liste mari de numere astfel încât poate fi aplicat un algoritm binar de căutare.

Până în ziua de astăzi, mulți informaticieni își dedică întreaga carieră pentru a creea algoritmi de sortare mai rapizi. Acest subdomeniu continuă să evolueze constant. De exemplu, în 2001, a fost inventat un algoritm nou numit Tim Sort care combină tot ce este mai bun din Merge și Insertion Sort (Tim Sort). Acest algoritm are complexitate $O(nlogn)$.

Sunt mai multe exemple, dar este important de precizat că algoritmii de sortare sunt fundamentali, mai ales pentru optimizarea timpului de rulare al programelor.

Planul acestei lucrări este să prezinte teoria și prezentarea formală a algoritmilor de sortare, urmată de reprezentarea computerizată a algoritmilor și, în final, a rezultatelor testelor de comparare a timpilor de rulare a acestora cu mai multe intrări.

Rezultatul anticipat este că algoritmii rapizi (Quick Sort, Merge Sort etc) vor ocupa primele locuri, iar cei lenți (Bubble Sort, Insertion Sort) vor ocupa locurile de jos.

2 Prezentarea algoritmilor de sortare

Acest capitol va prezenta o scurtă descriere teoretică cât și o scurtă explicație a fiecărui algoritm în parte.

2.1 Ce este un algoritm și timpul său execuție?

Un algoritm este o serie de pași ordonați care sunt executați de un program și utilizate pentru a rezolva anumite probleme. Fiecare pas este explicit și autonom.[3]

Timpul de executare al unui algoritm este definit ca numărul de pași necesari pentru ca algoritmul să fie executat, precum și timpul necesar pentru ca fiecare pas să fie executat. Pentru o listă de dimensiunea n, timpul de executare al unui algoritm de sortare este definit ca $T(n) = c1 * n + c2 * n + ...cn * n$, unde c(i) este costul unui pas, iar n este numărul de ori este executat. Pentru a converti în O(); care este definită ca complexitatea în cel mai rău caz a unui algoritm și indică, pentru o matrice de dimensiune n, cel mai rău timp posibil pe care îl poate lua algoritmul; cel mai semnificativ termen din T(n) luat fără constantă. Dacă $T(n) = n^2 * 3 + 9$, atunci complexitatea este $O(n^2)$.[1]

O complexitate medie este definită ca complexitatea la care ajunge de obicei un algoritm de cele mai multe ori și este notată cu $\theta()$

2.2 Bubble Sort

Bubble Sort este un algoritm simplu care schimbă continuu elementele adiacente dacă acestea nu sunt în ordine. Trece prin lista de numere și schimbă elementele între ele, repetânt ărocesul până când lista este sortată. Complexitatea este $O(n^2)$.[2]

2.3 Selection Sort

Selection Sort găsește valoarea minimă în lista de numere și o introduce în poziția corectă. Acest proces se repetă pentru următoarea valoare minimă. Complexitatea este $O(n^2)$.[2]

2.4 Insertion Sort

Insertion Sort preia un element din lista de numere și îl introduce în poziția sa corectă, la fel ca un pachet de cărți în viața reală. Insertion Sort are o complexitate de $O(n^2)$.[2]

2.5 Quick Sort

Quick Sort selectează un element pivot și sortează lista de numere astfel încât fiecare element din dreapta pivotului să fie mai mare decât acesta și fiecare

element din stânga este mai mic. Acest proces se repetă până când lista este sortată. Cel mai rău caz pentru Quick Sort este $O(n^2)$ când pivotul listei este primul sau ultimul element. De cele mai multe ori, totuși, are o medie de $\theta(n * logn)$. $O(n^2)$ este destul de rar pentru Quick Sort, așa că în scopul acestei lucrări, se va considera că Quick Sort este $O(n * logn)$.[2]

2.6 Merge Sort

Merge Sort împarte lista de numere în 2 subliste și face acest lucru până când listele împărțite sunt compuse doar din 1 element. După aceea, listele sunt îmbinate din nou împreună în ordine crescătoare. Are complexitate $O(n * logn)$.[2]

2.7 Heap Sort

Heap Sort creează un heap, care este un arbore binar special în care rădăcina este fie minim (min heap) fie maxim (max heap) și unde nodul părinte al unei frunze este fie mai mic (min heap) fie mai mare decât acesta (max heap).Lista este convertită într-un heap, sortând-o în acest proces. Acesta are complexitatea de $O(n * logn)$.[2]

2.8 Counting Sort

Counting Sort numără aparițiile unui element într-o listă separată, apoi fiecare element din lista de numărare este suma lui însuși și a elementelor anterioare. Din lista de numărare, este creată o listă de ieșiri în care fiecare element din originala listă este adăugat în funcție de valoarea sa din lista de numărare. Acest algoritm are o complexitate liniară $O(n)$.[2]

2.9 Radix Sort

Radix Sort poate sorta numai numere. Le sortează după cifrele lor, de la cea mai mică la cea mai semnificativă folosind sortarea de numărare. Are o complexitate liniară.[2]

2.10 Compararea algoritmilor dintr-o perspectivă teoretică

Bazat pe complexitățile lor:

- $O(n)$: Counting Sort, Radix Sort;
- $O(n * logn)$:Merge Sort, Quick Sort, Heap Sort;
- $O(n^2)$:Bubble Sort, Selection Sort, Insertion Sort;

Cu toate acestea, în lumea reală, nu este atât de simplu. Există anumite lucruri care trebuie luate în considerare înainte de a le compara.

În primul rând, mărimea listei de numere este un factor major în timpul lor de executare. De exemplu, o listă cu doar 3 elemente poate fi sortată mai rapid prin Bubble Sort decât prin Merge Sort, deoarece Merge Sort împarte mai întâi lista, decât o sortează, în timp ce Bubble Sort doar schimbă elementele și este gata.[1]
Insertion Sort este de obicei suficient de rapid pe liste de dimensiuni mici, fiind adesea opțiunea preferată pentru astfel de liste față de alți algoritmi (și anume Merge Sort).
După cum sa menționat anterior, Quick Sort nu este cel mai preferabil algoritm, deoarece este posibil să dureze la fel de mult ca n^2 pentru anumite tipuri de liste.
Merge Sort, Counting Sort precum și Radix Sort creează liste suplimentare care sunt utilizate pentru sortarea celei principale, care ocupă memoria RAM, astfel încât nu sunt eficiente în spațiu.
Există multe alte exemple, dar compararea algoritmilor de sortare se poate face în diferite moduri: compararea timpului de executare pentru anumite dimensiuni de liste, compararea stabilității, compararea cerințelor de memorie RAM și așa mai departe. În scopul acestei lucrări, se va compara timpul de executare a algoritmilor de sortare cu liste de dimensiuni diferite.

2.11 Compararea algoritmilor

Compararea timpului de executare al algoritmilor de sortare se face cel mai bine prin scrierea unui program care include toți algoritmii menționați.
Programul va avea o funcție pentru fiecare algoritm de sortare, precum și funcții pentru afișarea unei liste, schimbarea a două elemente și găsirea elementului maxim într-o listă. Elementele listei vor fi citite dintr-un fișier text și atribuite listei în ordinea apariției în fișierul menționat.
Programul va include și o bibliotecă care conține funcții folosite pentru a număra timpul de executare al algoritmilor în milisecunde. Aceste funcții numără doar perioada de timp din program atunci când se execută o funcție de algoritm de sortare: nu contorizează timpul necesar pentru a citi din fișier, afișând matricea sortată și altele.
Programul a fost scris în C++ și folosește alocarea dinamică a memoriei pentru a ține cont de dimensiuni mai mari pentru liste: utilizarea standard a variabilelor declarate pe stivă limitează matricea la o dimensiune de aproximativ 170 000 de elemente. Voi compara algoritmii de sortare pentru dimensiuni de maximum 1 milion de elemente.

3 Programul

Acest capitol prezintă implementarea programului folosit pentru compararea algoritmilor de sortare.Codul sursă al acestui program se găsește in acest link.

3.1 Descriereaa programului

După cum am menționat anterior, este un program C++ compus din mai multe funcții, una pentru fiecare algoritm de sortare împreună cu funcții utilitare pentru afișarea listei, schimbarea elementelor și găsirea elementului maxim. Funcția principală citește elemente dintr-un fișier text numai pentru citire (.txt) și construiește o matrice cu acele elemente. Biblioteca ¡time.h¿ este folosită pentru a număra timpul în milisecunde între două variabile ClockStart și ClockFinish, declarate înainte ca programul să înceapă citirea fișierul. Matricea este alocată dinamic folosind operatorul new și la sfârșitul programul se eliberează memoria folosind operatorul delete. Dimensiunea maximă din lista pentru acest experiment va fi de 1 milion.

4 Compararea algoritmului

În acest capitol, voi prezenta părerea mea cu privire la rezultate, precum și rezultatele reale ale comparației.

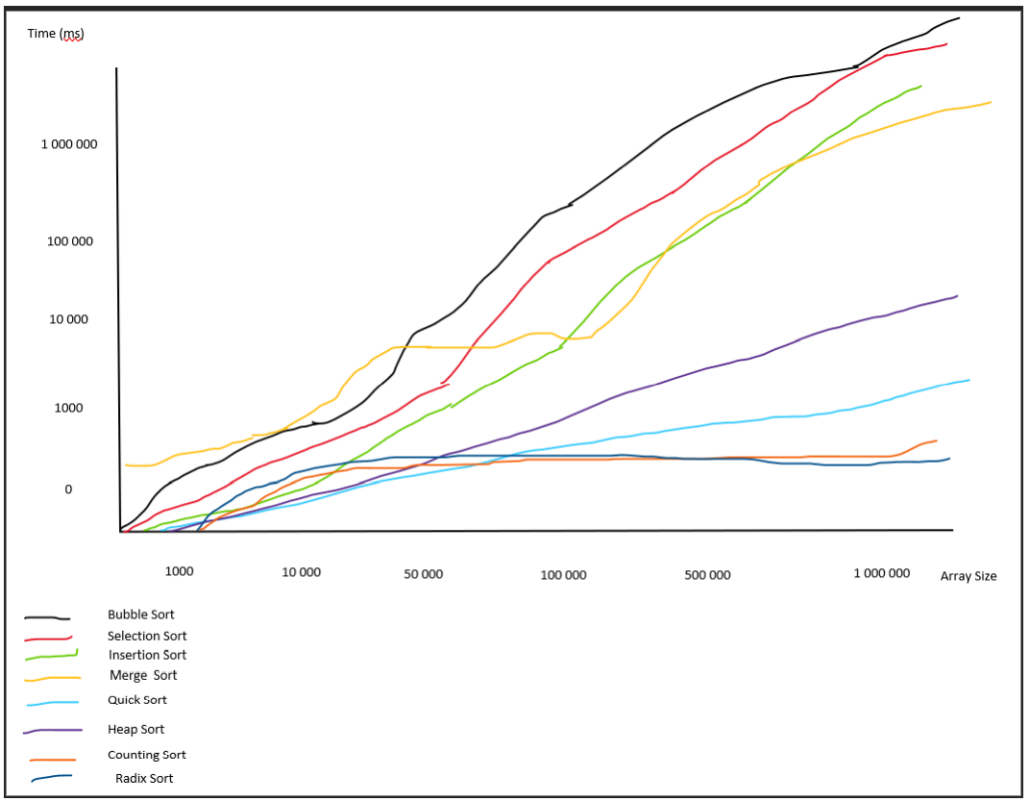
4.1 Părere personală

După cum am menționat în Capitolul 2, mă aștept ca algoritmii de sortare mai rapizi (Quick Sort, Merge Sort, Radix Sort,...) să ruleze în cel mai scurt timp, în timp ce algoritmii mai lenți (Bubble Sort, Selection Sort) să ruleze într-un timp semnificativ mai lung.
Voi începe cu o matrice de dimensiunea 1000, apoi voi trece la 10 000, 50 000, 100 000, 500 000 și în final 1 000 000. Elementele tabloului vor fi plasate aleatoriu și au posibilitatea de a se repeta. De asemenea, voi folosi doar numere întregi mai mari sau egale cu 0 din intervalul 0 - 1 000 000.
Următoarea parte a lucrării va conține rezultatele pentru diferite dimensiuni, precum și remarci și opinii personale asupra listei returnate. La sfârșitul acestui capitol, va exista un tabel și un grafic în care rezultatele tuturor testelor vor fi vizibile pentru mai ușor.

4.2 Rezultate

Algoritm de sortare/Dim. listă	1000	10 000	50 000	100 000	500 000	1 000 000
Bubble Sort	5 ms	627 ms	15 547 ms	61 922 ms	1 525 833 ms	1 oră
Selection Sort	1 ms	103 ms	2464 ms	9897 ms	234 562 ms	15 min
Insertion Sort	1 ms	50 ms	1532 ms	5448 ms	122 544 ms	10 min
Merge Sort	17 ms	170 ms	1061 ms	1914 ms	457 451 ms	16 min
Quick Sort	0 ms	2 ms	16 ms	33 ms	207 ms	900 ms
Heap Sort	1 ms	11 ms	91 ms	157 ms	948 ms	2 min
Counting Sort	6 ms	6 ms	7 ms	10 ms	14 ms	400 ms
Radix Sort	4 ms	5 ms	12 ms	19 ms	79 ms	600 ms

Table 1: Rezultatele execuției algoritmilor.



4.3 Observații si Interpretarea Rezultatelor

Având în vedere durata de timp pentru 500 000 de elemente pentru unii dintre algoritmi, am estimat aproximativ timpul necesar pentru 1 000 000 de elemente.
Ceea ce m-a surprins cel mai mult este că, cel puțin pentru liste de dimensiuni mici, cei mai lenți algoritmi (Bubble Sort, Selection Sort) sunt, de obicei, folosiți mai mult decât Merge Sort. Acesta nu pare să fie atât de rapidă pe cât este în teorie.
Pentru liste de dimensiuni mici, Counting Sort și Radix Sort sunt, de asemenea, relativ lente în comparație cu celelate. Cu toate acestea, pe măsură ce dimensiunea listei crește, acești doi algoritmi încep să urce pe primele locuri ca viteză, așa că per total sunt cei mai rapizi pentru dimensiuni mai mari.
Quick Sort este constant rapid, indiferent de dimensiune, clasându-se întotdeauna în top.
Bubble Sort devine îngrozitor de lent pe măsură ce dimensiunea crește, ducând până la o oră pentru o dimensiune de 1 000 000. Ceilalți algoritmi mai lenți (Insertion Sort și Selection Sort) durează mai mult cu cât dimensiunea listei este mai mare, dar creșterea timpilor lor de execuție nu este la fel de drastică ca și la Bubble Sort.
Heap Sort pare a fi o alegere solidă pentru dimensiuni mici, dar este depășit de Counting Sort și Radix Sort la dimensiuni mai mari, deși încă este destul de rapid.
Merge Sort pare să se comporte cel mai bine cu o matrice de dimensiuni medii, insă cu liste de dimensiuni mai mici și mai mari este, surprinzător, mai lent decât Insertion Sort și Selection Sort.
În ceea ce privește stabilitatea, Quick Sort și Heap Sort nu variază drastic la modificări de dimensiune. Ei sunt întotdeauna constant rapizi, în timp ce ceilalți algoritmi variază în funcție de mărimea listelor.
Graficul de mai jos, precum și tabelul oferă o prezentare generală a rezultatelor.

4.4 Concluzii

Pe baza rezultatelor pe care le-am obținut, aș spune că Quick Sort și Heap Sort sunt în general, cei mai rapizi și mai stabili algoritmi de sortare, Bubble Sort, Insertion Sort și Selection Sort sunt suficient de rapizi pentru liste de dimensiuni mici, în timp ce Radix și Counting Sort sunt cei mai rapizi pentru liste mari, iar Merge Sort este relativ rapidă pentru liste de dimensiuni medii.

5 Lucrari aferente

Pe lângă ceilalți colegi din facultate, care au avut aceeași sarcină de a scrie aceasta lucrare, există numeroase articole și lucrări științifice pe internet legate de această temă. Unele prezintă doar o comparație teoretică, în timp ce altele se concentrează mai mult pe comparații practice.
(Comparison of Sorting Algorithms) Acest site web analizează o comparație teoretică a algoritmilor de sortare în funcție de complexitatea și stabilitatea acestora.
(Comparison Sorting Visualization)Acest site web prezintă o animație a unei liste aleatorii fiind sortată după algoritmi diferiți. Nu înregistrează timpul in mod direct, dar o diferență între timpul de rulare al algoritmilor este destul de clară.
Desigur, există mai multe exemple de lucrari similare cu aceasta, atât pe internet cât și în cărți. Google Scholar, precum și Google sunt o resurse care ajută pe oricine să caute mai multe lucrări conexe și să valorifice aceste informații.
Comparând rezultatele mele cu standardul domeniului, aș spune pe lângă Merge Sort, care a fost puțin mai lent decât de obicei, ar trebui să fie suficient de precise. Era de așteptat acest lucru deoarece o mulțime de factori (performanța hardware, codul, instrumentele utilizate) joacă un rol în acuratețea rezultatelor.

6 Concluzii și posibile lucrări viitoare

Ca o concluzie a acestei lucrări, aș spune că compararea algoritmilor de sortare este importantă în principal pentru că îi învață pe programatori și pe alți oameni care algoritmi sunt cei mai eficienți în diferite situații. Acest lucru îi va ajuta să optimizeze codul pe care îl scriu pentru aplicații web, jocuri video, algoritmi de machine learning și așa mai departe.
În ceea ce privesc rezultatele mele, acestea se potrivesc, în mare parte, cu timpul de rulare teoretic prezentat în capitolul 2, totuși există cu siguranță fluctuații și diferențe în rezultatele practice pe care le-am obţinut.
Pe viitor, s-ar putea să adaug mai mulți algoritmi de sortare la test și aș putea mării dimensiunile listelor. În afară de acestea, aș putea varia și tipul de date folosit, deoarece acest experiment s-a concentrat doar pe numere întregi pozitive. Indicat ar fi sa implementez propriul meu generator de elemente, deoarece am folosit un generator online care era limitat la un număr maxim de 10 000 de elemente.

Bibliografie

[1] Saleh Abdel-Hafeez and Ann Gordon-Ross. An efficient o (n) comparison-free sorting algorithm. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(6):1930–1942, 2017.

[2] Aditya Dev Mishra and Deepak Garg. Selection of best sorting algorithm. *International Journal of intelligent information Processing*, 2(2):363–368, 2008.

[3] H Cormen Thomas. -introduction to algorithms, 1990.