Problema Sortării

Georgescu Andrei Călin: 324CA[1]

Universitatea Politehnică București, Facultatea de Automatică și Calculatoare
http://acs.pub.ro

13 Decembrie 2018

Abstract. Problema sortării stă la baza multor algoritmi complecși din Șttința Calculatoarelor. Prezenta lucrare analizează performanțele a trei algoritmi de sortare des folosiți în practică (QuickSort, MergeSort, HeapSort), din punct de vedere al timpului de execuție și a memoriei utilizate, pe diferite segmente de date. Scopul lucrării este determinarea cazului optim de folosire pentru fiecare algoritm în parte.

Keywords: QuickSort · MergeSort · HeapSort · Complexitate · Eficiență · Caz optim de utilizare.

1 Introducere

1.1 Descrierea Problemei Rezolvate

Fiind dat un vector de N elemente (cu numere întregi), dispuse intr-o ordine aleatoare, se dorește sortarea acestora într-o ordine crescătoare. Lucrarea prezentă se axează pe compararea eficienței a trei algoritmi prin comparare (QuickSort, MergeSort, HeapSort).

Pentru testare s-au ales vectori de numere întregi de diferite lungimi și cu elemente generate aleator.

1.2 Exemple de aplicații practice pentru problema aleasă

Soratarea este unul dintre aspectele fundamentale ale științei calculatoarelor. Multe dintre probleme actuale pot fi reduse la o problema de sortare.

Exemplu 1 (Medianul și Statistici de Ordine) Se pune problema determinării medianului dintr-o serie de chei date(o valoare cu proprietatea că jumatate dintre cheile date sunt mai mici, iar cealaltă jumătate este mai mare decât valoarea căutată). Această operație este una comună în statistică și în diverse alte aplicații de procesare de date. Este facilă rezolvarea acestei probleme într-un timp liniar sortând mai întâi lista de chei date.

Exemplu 2 (Păstrarea și căutarea de informații) Instituțiile financiare și comerciale își organizează informațiie despre clienți sortate în funcție de un anumit criteriu (nume, număr, tranzacțiile pot fi păstrate în funcție de dată sau locul efectuării etc.). Păstrând aceste date într-o manieră sortată facilitează căutarea și accesarea informațiilor dorite într-un timp cât mai scurt.

Exemplu 3 (Computații numerice) Diverse operații științifice își pun problema determinarii unui rezultat cu o precizie cât mai mare. Acuratețea este importantă atunci când se realizează un număr extrem de mare de operații cu valori estimative (reprezentarea cu virgulă mobilă a numerelor reale). Unii algoritmi numerici necesită folosirea unor cozi de prioritate și algoritmi de sortare pentru a putea controla acuratețea în calcule.

1.3 Specificarea soluțiilor alese

În alegerea soluțiilor s-au luat în considerarea factori precum ușurința implementării, eficiența de timp și cea de spațiu a algoritmilor. Așadar s-au selectat urmatorii algoritmi:

- QuickSort;
- MergeSort;
- HeapSort.

1.4 Specificarea criteriilor de evaluare alese pentru validarea soluțiilor

Criteriile de evaluare ale soluțiilor alese se bazează pe factorii luați în considerare în alegerea soluțiilor. Seturile de date au fost concepute astfel încât să trateze cazul favorabil, defavorabil și mediu de utilizare pentru fiecare algoritm în parte. Așadar, pentru validarea soluțiilor s-au urmărit diverși indicatori:

- Sortarea în mod corect a datelor de intrare;
- Durata de execuție a fiecărui algoritm pentru diversele seturi de date;
- Spațiul folosit de fiecare algoritm pentru sortarea corectă a seturilor de date;
- Ușurința de implementare a algoritmilor.

2 Prezentarea soluțiilor

2.1 Descrierea modului în care funcționează algoritmii aleși

În continuare se va detalia funcționarea fiecărui algoritm de sortare ales în parte.

QuickSort. Este un algoritm folosit des în practică datorită implementării sale facile. Diferențele de performanță se pot aduce schimbând funcția de partiționare, respectiv alegerea pivotului. Pentru sortarea unui vector de N elemente A[p...r] se procedează astfel:

- **Partiționare:** Se partiționează vectorul A[p...r] în doi subvectori A[p...q-1] și A[q+1...r] astfel încât oricare element din A[p...q-1] este mai mic sau egal ca A[q], care, la rândul său, este mai mic sau egal ca oricare element din A[q+1...r]. Se reține indicele q;
- **Sortare:** Se sortează subvectorii A[p...q-1] și A[q+1...r] prin apeluri recursive ale algoritmjului quicksort.

Observație: subvectorii sunt sortați in - place, deci nu este nevoie de lucru suplimentar pentru sortarea vectorului A[p...r].

MergeSort. La fel ca și QuickSort, MergeSort urmărește îndeaproape paradigma Divide et impera. Pașii algoritmului sunt urmatorii:

- **Divide:** Împărțim vectorul de N elemente ce se dorește a fi sortat în doi vectori de câte N/2 elemente fiecare;
- **Impera:** Sortăm cei doi subvectori folosind recursiv mergeSort;
- **Combinarea soluțiilor:** Îmbinăm cei doi vectori pentru a produce răspunsul așteptat.

Observație: Recursivitatea se va opri atunci când subvectorul care trebuie sortat ajunge de lungime 1, caz în care nu mai este necesară nicio operație de sortare, din moment ce orice secvență de un element se consideră deja sortată.

Algoritmul se bazeaza pe pasul de combinare a solutiilor. Se presupune că p este indicele de pornire a sortării, q indicele curent și r indicele de stop care se află în relația $p \leqslant q \leqslant r$. Considerăm vectorul de sortat ca fiind B[N].

Presupunând că subvectorii B[p...q] și B[q+1...r] sunt sortați, acest algoritm va calcula rezultatul în vectorul B[p....r], suprascriind vectorul inițial.

HeapSort. Algortimuul de HeapSort se bazează pe crearea unui MaxHeap din care se va extrage la fiecare pas rădăcina (numărul cel mai mare din setul de date). Presupunem că vectprul de sortat este A[N]. Pașii algoritmului sunt următorii:

- **Partea principală:** Rădăcina heapului va fi A[1], iar dacă vom considera i indexul unui nod, indicii părintelui său, PARENT(i), al copilului stâng LEFT(i) și al copilului drept RIGHT(i) pot fi aflați cu ușurință, astfel:

  PARENT(i) = i/2;
  LEFT(i) = 2 * i;
  RIGHT(i) = 2 * i + 1.

- **Partea de Heapify:** Se va verifica proprietatea de max-heap:
  $A[PARENT(i)] \leqslant A[i]$. Inputul pentru funcția de MaxHeap va fi un vector A și un index i. Când funcția va fi apelată, se presupune că arborii cu

rădăcinile în LEFT(i) și RIGHT(i) sunt maxheapuri, dar C[i] poate fi mai mic decât copiii săi. RepairHeap va coborî valoarea A[i] astfel încât să se mențină proprietatea de maxheap. Pseudocodul funcției:

```
MAX-HEAPIFY(A, i)
 1  l ← LEFT(i)
 2  r ← RIGHT(i)
 3  if l ≤ heap-size[A] and A[l] > A[i]
 4     then largest ← l
 5     else largest ← i
 6  if r ≤ heap-size[A] and A[r] > A[largest]
 7     then largest ← r
 8  if largest ≠ i
 9     then exchange A[i] ↔ A[largest]
10          MAX-HEAPIFY(A, largest)
```

Fig. 1. Pseudocodul funcției de MaxHeap [1].

– **Partea de construcție a Heapului:** Folosind funcția de MaxHeap, putem construi un heap ținând cont că N = length[A]. Pseudocodul funcției arată astfel:

```
BUILD-MAX-HEAP(A)
1  heap-size[A] ← length[A]
2  for i ← ⌊length[A]/2⌋ downto 1
3       do MAX-HEAPIFY(A, i)
```

Fig. 2. Pseudocodul funcției de construcție a heapului [1].

– **Partea de obținere a vectorului sortat:** Algoritmul începe prin construirea unui maxHeap, din care se extrage rădăcina (care va fi mereu elementul maxim din vector/maxHeap) și se interschimbă cu A[N]. Apoi vom elimina nodul rădăcină prin decrementarea dimensiunii heapului și vom repara heapul folosind funcția de maxHeap. Pseudocodul va arăta astfel:

```
HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i ← length[A] downto 2
3      do exchange A[1] ↔ A[i]
4         heap-size[A] ← heap-size[A] − 1
5         MAX-HEAPIFY(A, 1)
```

Fig. 3. Pseudocodul funcției de HeapSort [1].

2.2 Analiza complexității algortmilor

QuickSort: Se determină trei cazuri de utilizare pentru algoritmul de QuickSort: Cazul defavorabil de partiționare, cazul favorabil de partiționare și cazul mediu. În continuare vom analiza complexitatea algoritmului pentru fiecare dintre cazurile prezentate.

- **Cazul defavorabil** Acesta apare când partiționare produce doi subvectori, unul de N - 1 elemente și altul cu 0 elemente. Se presupune că acest lucru se întâmplă pentru fiecare apel recursiv al funcției de partiționare. Partiționarea se realizează în $\mathcal{O}(N)$. Apelul recursiv pentru subvectorul cu 0 elemente se realizeaza în T(0) = $\mathcal{O}(1)$, iar recurența pentru timpul de rulare va fi:
  T(N) = T(N - 1) + T(0) + $\mathcal{O}(N)$ = T(N-1) + $\mathcal{O}(N)$
  Dacă vom calcula suma tuturor apelurilor vom obține o complexitate totală de $\mathcal{O}(n^2)$. Acest lucru se obține atunci când vectorul de input este deja sortat.
- **Cazul favorabil** Acesta apare când vectorul de input este cât se poate de bine împărțit. Funcția de partiționare va produce doi subvectori, fiecare de N/2 elemente. În acest caz recurența de timp devine:
  $T(N) \leqslant 2 * T(N/2) + \mathcal{O}(N)$, care, aplicând terema Master, va rezulta îbtr-o complexitate totală T(N) = $\mathcal{O}(NlogN)$
- **Cazul Mediu** Presupunând o împărțire de 8:2 a vectorului de input, algoritmul de QuickSort va avea o complexitate
  $T(N) \leqslant) T(8 * N/10) + T(2 * N/10) + an$, unde a este o constantă. Recursivitatea va avea așadar o complexitate de $\mathcal{O}(NlognN$ Așadar, complexitatea algoritmului în acest caz se aproprie de cea din cazul favorabil de folosire și se află în aceeași clasă de complexitate[2].

MergeSort: Pentru determinarea complexității presupunem că vectorul de intrare are un număr N de elemente, N fiind o putere a lui 2. Fiecare pas recursiv va crea subvectori de N/2 elemente. Astfel obținem complexitatea pentru T(N). MergeSort pentru un element are timp de execuție constant. Pentru N > 1 avem:
Divide: D(N) = $\mathcal{O}(1)$;

Impera: I(N) = 2 * T(N/2);
Combinarea soluțiilor: C(N) = $\mathcal{O}(N)$.
Astfel, pentru N > 1 elemente vom avea o complexitate totală T(N) = 2 * T(N/2) + $\mathcal{O}(N)$. Folosind teorema master obținem că T(N) = $\mathcal{O}(NlogN)$

HeapSort Pentru determinarea complexității vom analiza fiecare funcție a algoritmului în parte.
Max-Heapify: Funcția rulează într-un timp de $\mathcal{O}(logN)$;
Build-Max-Heap: Rulează într-un timp liniar;
Heapsort: Rulează într-un timp de $\mathcal{O}(NlogN)$ și produce vectorul sortat.

Așadar, complexitatea algoritmului de Heapsort este $\mathcal{O}(NlogN)$.

2.3 Principalele avantaje și dezavantaje pentru soluțiile luate în considerare

În cele ce urmează vom analiza fiecare algoritm în parte și îi vom prezenta avantajele, respectiv dezavantajele.

QuickSort

- **Avantaje:** Principalul avantaj al algoritmului este lucrul cu liste mari de elemente. Pentru faptul că sortează elementele in - place, nu este nevoie nici de memorie adițională.
- **Dezavantaje:** Un dezavantaj al acestui algoritm este că are un caz defavorabil care se soluționează într-un timp comparabil cu cazul mediu de utilizare pentru BubbleSort, InsertionSort sau SelectionSort. Un alt dezavantaj este că produce o sortare instabilă, datorită inversării elementelor în pasul de partiționare.

MergeSort

- **Avantaje:** Un avantaj al algoritmului este că, dată fiind implementarea algoritmului, nu este necesar ca toate datele ce trebuiesc sortate să se afle în aceeași zonă de memorie. Principalul avantaj, însă, este că are un timp de rulare de $\mathcal{O}(NlogN)$, chiar și în cazul defavorabil, ceea ce este optim. De asemenea, variațiile acestei sortări (ex. Tim Sort) aduc îmbunătățiri semnificative de performanță[2].
- **Dezavantaje:** Principalul dezvantaj este că sunt necesare mai multe operații de copiere a datelor din vector și crearea de subvectori. Algoritmul este, totodată, ineficient din punct de vedere al spațiului folosit, tocmai din acest caz.

HeapSort

- **Avantaje:** Rulează într-un timp de $\mathcal{O}(NlogN)$ și poate fi modificat cu ușurință pentru a permite implementarea sortării in - place.
- **Dezavantaje:** Algoritmul nu este la fel de rapid ca algoritmii ce se bazează pe comparare. Un alt dezavantaj este că algoritmul nu produce o sortare stabilă.

3 Evaluare

3.1 Descrierea modalității de construire a setului de date folosite pentru validare

Pentru generarea seturilor de date s-a folosit un program .cpp generator, care cu ajutorul Makefile-ului generează o nouă pereche de fișiere .in și .out. Acestea se pot redenumi și plasa în folderele corespunzătoare. Fișierul .cpp conține cod care poate fi comentat sau decomentat pentru a genera teste care să îndeplinească oricare dintre cazurile de evaluare a performanței descrise.
Seturile de date au fost generate astfel încât să testeze cazul favorabil, defavorabil și mediu de utilizare pentru fiecare algoritm ales în parte. Datele de intrare sunt N, dimensiunea vectorilor și pe următoarea linie cele N elemente ale vectorilor. Seturile de date conțin: vectori sortați crescător, vector sortați descrescător, vectori parțial sortați și vectori cu date complet aleatorii. Pe lângă acestea, a fost creat un test separat cu un număr foarte mare de elemente.

Observație: În testele de intrare se vor afla și elemente duplicat.

3.2 Menționarea specificațiilor sistemului de calcul pe care au fost rulate testele

- **Procesor: System:2,16 %, User: 4%, Idle: 93,84%**
- **Memorie RAM disponibilă: 20GB**

3.3 Ilustrarea, folosind grafice/tabele, a rezultatelor evaluării soluțiilor pe setul de teste.

Timpii au fost măsurați la rulare cu o precizie de 6 zecimale.

Table 1. Tabel conținând rezultatul testelor pentru fiecare algoritm în parte pe setul default de date .

Numărul testului	QuickSort	MergeSort	HeapSort
0	0.000192	0.000147	0.000149
1	0.000211	0.000100	0.000098
2	0.000145	0.000134	0.000133
3	0.000455	0.000394	0.000390
4	0.002905	0.000412	0.000398
5	0.000664	0.000179	0.000161
6	0.001197	0.000999	0.001006
7	0.002699	0.002312	0.002327

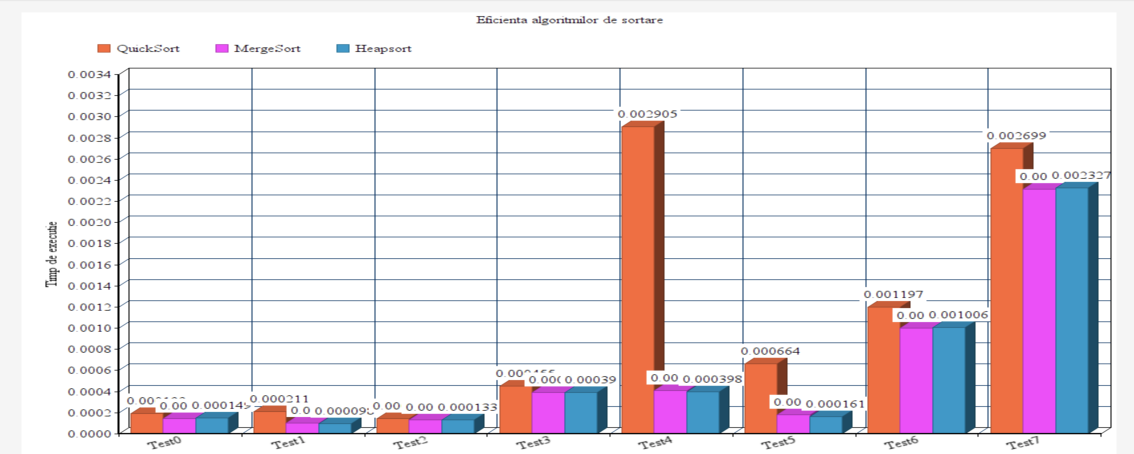

Fig. 4. Graficul eficienței algoritmilor de sortare

3.4 Prezentarea succintă a valorilor obținute pe teste.

Valorile obținute pe teste au fost cele așteptate. Pe lângă cele 8 teste default, vom analiza algoritmii și pe testele cu numere foarte mari din folderul othertests.

Se observă că algoritmul de QuickSort este mai eficient ca celelalte pe seturi de date mari, așa cum era de așteptat.

Table 2. Tabel conținând rezultatul pentru testul pe un vector cu număr mare de elemente.

Numele Testului	QuickSort	MergeSort	HeapSort
largeVector0	0.03246300	0.03295300	0.03401100
largeVector1	0.04535400	0.05391700	0.05707900

4 Concluzii

În urma acestei lucrări se poate concluziona că există un caz optim de utilizare pentru fiecare algoritm. Astfel, pentru seturi mari de date se va alege algoritmul de QuickSort, însă cu îmbunătățiri (la nivelul alegerii pivotului), pentru a eficientiza și mai mult operația de sortare. Pentru situațiile în care dorim sortarea unor date care nu se află în aceeași zonă de memorie vom utiliza algoritmul de MergeSort, iar pentru situațiile în care dimensiunea datelor de intrare este variabilă se va folosi algoritmul de HeapSort, care are complexitate constantă.

References

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. Introduction to Algorithms, 3rd Edition, The MIT Press
2. Edosomwan Joseph. Sorting Algorithm: Analysis and Comparison Perfor- mance, Lambert Publishing, 19 Jul 2012