Comparație teoretică și experimentală a algoritmilor de sortare

Iuliana Călina Mihali
Studentă la Informatică
Facultatea de Matematică și Informatică
Universitatea de Vest Timișoara, România
Email: iuliana.mihali02@e-uvt.ro

11 mai 2024

Rezumat

Lucrarea prezintă o comparație și analiză teoretică și experimentală a 5 algoritmi de sortare cunoscuți: Quicksort, MergeSort, BubbleSort, SelectionSort și InsertionSort.

În următoarele secțiuni vor fi prezentate mai multe tipuri de date de test cu care acești algoritmi au fost testați. Este important de menționat că rezultatele experimentale prezentate pot varia față de cele din alte studii sau diferite implementări, în funcție de specificul implementării fiecărui algoritm și de capacitatea de procesare a calculatorului utilizat în testări.

Aceste variații sunt importante pentru înțelegerea completă a performanței fiecărui algoritm și accentuează importanța considerării contextului tehnic în evaluarea lor comparativă. Ca urmare a testelor in lucrare sunt menționate concluziile asupra algoritmilor pe baza creșterii exponențiale a timpului de execuție în urma datelor de dimensiune mai mare.

Cuprins

Listă de tabele

Fragmente de cod

1 Introducere

În programare, sunt folosiți numeroși algoritmi de sortare, în această lucrare, vom analiza cinci dintre cei mai frecvent utilizați algoritmi de sortare care se bazează pe compararea elementelor. Mai jos vom enumera cei cinci algorimti care o să fie analizați:

- Algoritmi de sortare eficienți cu complexitatea - $O(n * log(n))$:
  - **Quicksort** *(sortare rapidă)*
  - **Merge sort** *(sortare prin interclasare)*
- Algoritmi de sortare neeficienți cu complexitatea - $O(n^2)$:
  - **Bubble sort** *(sortarea bulelor)*
  - **Insertion sort** *(sortare prin inserție)*
  - **Selection sort** *(sortare prin selecție)*

1.1 Motivație

Unii dintre cei mai folosiți algoritmi sunt cei de sortare care au aplicabilitate în mai multe probleme, astfel orice programator trece prin procesul de cunoaștere a cel puțin unuia dintre algoritmii de sortare. Cu cât cunoaștem un algoritm de sortare mai eficient, cu atât complexitatea problemei este mult mai scăzută, ceea ce face ca programul să scoată un timp foarte optim de execuție.

Prin urmare, pe măsură ce setul de date este format dintr-un volum mare, este necesară alegerea unui algoritm de sortare potrivit pentru a menține performanța destul de ridicată a programului nostru, deoarece de multe ori sortarea datelor într-un sistem poate fi destul de critică asupra eficienței acestuia.

1.2 Exemple de probleme

Să considerăm un caz real a unui joc competitiv unde mii de jucători participă la un turnament, la finalul fiecărui meci scorurile trebuie actualizate iar clasamentul trebuie reordonat. Folosirea unui algoritm precum bubble sort în acest caz ar duce la un timp de execuție foarte mare, ceea ce ar fi foarte deranjant pentru jucătorii care așteaptă scorul după terminarea meciului. În schimb, daca am folosi quicksort, clasamentul ar fi actualizat în câteva secunde, neafectând performanța jocului.

Un alt exemplu potrivit pentru folosirea unui algoritm de sortare cu un timp de execuție mai ridicat față de un algortim de execuție mai scăzut este cazul unei biblioteci mici în care catalogul digital conține aproximativ 500-1000 de cărți. Sortarea prin selecție este mai potrivită decât quicksort datorită faptului că diferența de eficiență între algoritmi nu este atât de evidentă pentru un set de date mai mic, iar simplificarea procesului de sortare a selecției ar putea fi mai avantajoasă.

Totodată sortarea prin selecție poate fi folosită într-o problemă reală unde este necesar sortarea a unui număr de k elemente dintr-un număr de n elemente unde n este un număr extrem de mare, astfel sortarea prin selecție ar putea fi mult mai eficientă decât quicksort-ul deoarece sortează doar primele k elemente iar celelalte n-k elemente rămân neordonate, iar quicksort-ul sortează toate elementele astfel ajungând la un timp mai ridicat de execuție. Acest caz se aplică doar în momentul în care k nu este un număr foarte mare.

În continuarea lucrării vor fi prezentate analiza teoretică și descrierea algoritmilor de sortare în secțiunea 2, mai departe vom găsi metode de testare a algoritmilor în secțiunea 3, iar rezultatele experimentale vor putea fi găsite in secțiunea 4. În urma acestora, concluziile si direcțiile viitoare vor rezulta în secțiunea 6.

2 Analiza teoretică a problemei și a soluțiilor

2.1 Preliminarii

În următoarea secțiune pentru a înțelege partea teoretică vom prezenta câteva aspecte pe care le vom folosi:

- N: numărul de elemente din șir;
- T(k): numărul de comparații efectuate la al k-lea pas;
- O (Big-O): complexitatea algoritmului în cel mai prost caz;
- Θ (Big-Theta): complexitatea algoritmului în cazul mediu.
- Ω (Big-Omega): complexitatea algoritmului în cel mai bun caz;
- Memorie: spațiul ocupat de algoritmi

2.2 Comparație și analiza teoretică

Pentru a putea compara algoritmii de sortare enumerați mai sus, vom încerca în următoarea parte să arătăm performanța acestora prin exemplificarea complexității fiecărui algoritm.

- **Quicksort:** Este un algoritm eficient de sortare, iar numărul de comparații necesar pentru a sorta n elemente poate fi exprimat prin relația de recurență $T(n) = T(k) + T(n-k-1) + \Theta(n)$. În cel mai nefavorabil caz, recurența devine: $T(n) = T(n-1) + \Theta(n)$, ceea ce conduce la o complexitate de timp de $O(n^2)$. În cel mai favorabil caz, avem $T(n) = 2*T(n/2) + \Theta(n)$, folosind Teorema Master, complexitatea este $\Omega(n * log(n))$.
- **Merge sort:** Pentru sortarea a n elemente, numărul total de comparații poate fi exprimat prin următoarea relație de recurență: $T(n) = 2*T(n/2)$

$+ \Theta(n)$. Acest model de recurență indică o complexitate de timp de $O(n * log(n))$.

- **Bubble sort:** Acest algoritm efectuează, la fiecare iterație, (n - i) comparații, unde i reprezintă numărul iterației curente. Prin urmare, numărul total de comparații realizate este sumat ca $\sum_{i=1}^{n} n - i = \frac{n(n-1)}{2}$. Complexitatea calculată în urma acestui proces este $\Theta(n^2)$.
- **Insertion sort:** La fiecare pas i, se efectuează (i - 1) comparații. Astfel numărul total de comparații va fi $\sum_{i=2}^{n} i - 1 = \frac{n(n-1)}{2}$. Complexitatea rezultată este $\Theta(n^2)$.
- **Selection sort:** În cadrul fiecărei iterații i, algoritmul efectuează (n-i) comparații. Totalul comparațiilor executate de-a lungul tuturor iterațiilor este reprezentat de suma $\sum_{i=1}^{n} n - i = \frac{n(n-1)}{2}$. Acest calcul duce la o complexitate algoritmică de $\Theta(n^2)$.

În următorul tabel sunt prezentați fiecare dintre algoritmii de mai sus împreuna cu timpul de execuție și spațiul ocupat pentru toate cele trei cazuri:

Algoritm	Cazul defavorabil	Cazul favorabil	Cazul mediu	Memorie
Quicksort	$O(n * log(n))$	$\Omega(n * log(n))$	$O(n^2)$	$O(n * log(n))$
Merge sort	$O(n * log(n))$	$\Omega(n * log(n))$	$\Theta(n * log(n))$	$O(n)$
Bubble sort	$O(n^2)$	$\Omega(n)$	$\Theta(n^2)$	$O(1)$
Insertion sort	$O(n^2)$	$\Omega(n)$	$\Theta(n^2)$	$O(1)$
Selection sort	$O(n^2)$	$\Omega(n^2)$	$\Theta(n^2)$	$O(1)$

Tabela 1: Complexitatea (timp și spațiu de memorie).

2.3 Soluția algoritmică

Bubble sort: La început, a fost menționat ca *Sorting by exchange* (Sortare prin interschimbare) în lucrările lui [Fri56], și ulterior ca *Exchange Sorting* în [Joh60]. Termenul *Bubble sort* a fost utilizat pentru prima dată de Kenneth E. Iverson în [Ive62].
Algoritmul Bubble Sort este o metodă simplă de sortare a unei liste de elemente, ușor de implementat, dar este destul de lent pentru liste mari comparativ cu alți algoritmi, acesta având complexitatea în cazul defavorabil de $O(n^2)$, dar pentru o listă aproape sortată poate fi util.
Cum funcționează?
Se parcurge tabloul de elemente, fiecare element este comparat cu vecinul său, dacă se găsesc 2 elemente neordonate atunci se vor interschimba, acest proces se va repeta atât timp cât nu vor mai fi găsite elemente neordonate.

Insertion sort: A fost inițial menționat de John Mauchly în anul 1946, în cadrul primei discuții publicate privind sortarea computațională, prezentată în lucrarea lui Donald E. Knuth, conform referinței [Knu98]. Pentru tablourile cu mai puține elemente, sortarea prin inserție este o metodă cunoscută de sortare, funcționează în felul următor:

1. Se alege un element dintr-o listă nesortată.
2. Se face o comparație între acest element și celelalte elemente din lista sortată, mai apoi se inserează pe poziția corespunzătoare.
3. Procesul este repetat pentru fiecare element din lista nesortată până când toată lista este sortată.

Așadar, pentru seturi mici de date sau deja aproape sortate, acest algoritm este eficient deoarece nu necesită multe comparații și interschimbări de elemente, dar pentru seturi mari de date complexitatea in cazul defavorabil este de $O(n^2)$, devenind foarte lent pentru liste mari sau neordonate.

Selection sort: Selection sort este un algoritm simplu de sortare, eficient pentru un set de date mici.
Funcționează în modul următor:
se parcurge lista, căutăm la fiecare pas elementul minim din sublista elementelor nesortate(ne parcurse) și se inserează pe poziția curentă din sublista sortată (parcursă). Metoda a fost prezentată în cartea lui Donald E. Knuth pentru prima dată în 1998, conform [Knu98].

Quicksort: Este un algoritm de sortare rapid și eficient, fiind și cel mai popular, bazat pe conceptul Divide et Impera.
Acest algoritm conceput în 1959 de către Tony Hoare, a fost dezvoltat în timpul unei vizite în calitate de student la Universitatea de Stat din Moscova, iar in 1961 a prezentat principiile și funcționarea acestuia într-un mod mai abstract, vezi [Hoa61].
Funcționează astfel:

1. Se alege o valoare arbitrară din tablou (prima, mediana, ultima sau aleatoare), aceasta reprezentând valoarea pivotului.
2. Elementele tabloului sunt rearanjate astfel: elementele mai mici decât pivotul sunt plasate în prima parte a tabloului, iar cele mai mari decât pivotul sunt mutate în partea a doua a tabloului.
3. Se aplică din nou acești pași pe cele două subtablouri formate.
4. Se concatenează subtablourile sortate.

Merge sort: Metoda de sortare prin interclasare, cunoscută sub numele de merge sort, a fost propusă ca una dintre primele soluții algoritmice pentru sortarea eficientă a datelor pe calculatoare. Inițial concepută de John Von Neumann în 1945, această tehnică a fost subiectul unei analize detaliate și discuții într-un raport tehnic redactat de Goldstine și Neumann. Mai multe informații pot fi găsite în referința [KT97]. Este un algoritm de sortare bazat pe conceptul Divide et

Impera, eficient fiind pentru seturi de date mari datorită complexității sale in cazul defavorabil de $O(n * log(n))$.
Funcționează astfel:

1. Găsește mijlocul listei și împarte lista inițială in 2 subliste, apelând algoritmul pe prima sub listă, pe a doua sub listă, până când fiecare sublistă conține un singur element.
2. Îmbină sublistele sortate pentru formarea listei finale sortate.

3 Modelare și implementare

Pentru a testa fiecare algoritm în parte au fost create teste pentru elemente generate aleator, elemente deja sortate, elemente sortate descrescător și elemente sortate pe jumătate și jumătate a șirului generată random. Șirurile au fost generate de ordinul 10^n, iar tot codul a fost scris în limbajul de programare python3. Pentru a testa timpul fiecărui algoritm a fost folosit un timer care era pornit înaintea algoritmului și se termina cand se termina execuția algoritmului. Mai jos se află codul pentru testarea timpului:

```
n = 5
array_large = [random.randint(1, 10**n) for _ in range(10**n)]

start_time = time.time()
# our algorithm
end_time = time.time()

execution_time_large = end_time - start_time
print(execution_time_large)
```

Fragment de cod 1: Test part

În următoarea parte mai jos se află codul pentru fiecare algoritm:

- Quick sort:

```
def quickSort(array, low, high):
    if low < high:
        pivot_index = partition(array, low, high)
        quickSort(array, low, pivot_index - 1)
        quickSort(array, pivot_index + 1, high)

def partition(array, low, high):
    pivot = array[high]
    i = low - 1
    for j in range(low, high):
        if array[j] <= pivot:
            i += 1
            array[i], array[j] = array[j], array[i]
    array[i + 1], array[high] = array[high], array[i + 1]
    return i + 1
```

Fragment de cod 2: QuickSort

- Merge sort:

```python
def mergeSort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]
        mergeSort(L)
        mergeSort(R)
        i = j = k = 0

        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
            k += 1
```

Fragment de cod 3: Merge sort

- Bubble sort:

```python
def bubbleSort(array):
    for i in range(len(array)):
        for j in range(0, len(array) - i - 1):
            if array[j] > array[j + 1]:
                temp = array[j]
                array[j] = array[j + 1]
                array[j + 1] = temp
```

Fragment de cod 4: Bubble sort

- Insertion sort:

```python
def insertionSort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
```

Fragment de cod 5: Insertion sort

- Selection sort:

```python
def selectionSort(arr):
    for i in range(len(arr)):
        min_idx = i
        for j in range(i+1, len(arr)):
            if arr[j] < arr[min_idx]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]
```

Fragment de cod 6: Selection sort

Lucrarea impreuna cu codul se vor gasi la următoarea adresă:
`https://github.com/iulianamihali/mpi_algoritmi_de_sortare.git`

4 Studiu de caz / experiment

Calculatorul unde au fost efectuate testările are urmatătoarele specificații:

- Procesor: Apple Silicon (M1 Pro)
- Ram: 16GB
- OS: macOS Sonoma
- IDE: Clion
- Language: Python3

În urmatoarea partea vor fi prezentate rezultatele pentru 4 tipuri de lungimi alese ale sirurilor de numere: $10^3, 10^4, 10^5$ și 10^7 împreună cu cele 4 tipuri de siruri alese menționate mai sus

10^3	Șir sortat	Șir descrescător	Șir jumătate sortat	Șir random
Quick sort	0.00303s	0.00266s	0.00278s	0.00291s
Merge sort	0.00099s	0.00140s	0.00100s	0.00200s
Bubble sort	0.01994s	0.04687	0.03288s	0.03489s
Insertion sort	0.00099	0.03451	0.01195	0.01487s
Selection sort	0.01694	0.01787	0.02276	0.01688s

10^4	Șir sortat	Șir descrescător	Șir jumătate sortat	Șir random
Quick sort	0.02435s	0.02256s	0.02085s	0.01177s
Merge sort	0.01718s	0.01941s	0.01977s	0.02231s
Bubble sort	2.07905s	4.80773s	2.44828s	3.63938s
Insertion sort	0.00106s	3.32268s	0.42773s	1.72255s
Selection sort	1.62207s	1.74649s	1.65824	1.65050s

10^5	Șir sortat	Șir descrescător	Șir jumătate sortat	Șir random
Quick sort	0.21797s	0.22796s	0.22866s	0.12947s
Merge sort	0.182973s	0.18360s	0.19970s	0.22224s
Bubble sort	206.25111s	483.59093s	246.57493s	370.33885s
Insertion sort	0.01101s	334.16685s	42.53397s	168.56546s
Selection sort	161.77508s	173.24954s	164.78961s	163.25679s

10^7	Șir sortat	Șir descrescător	Șir jumătate sortat	Șir random
Quick sort	2.44811s	2.43982s	2.44433s	1.72207s
Merge sort	2.07722s	2.09424s	2.43302s	2.63530s
Bubble sort	Prea mare	Prea mare	Prea mare	Prea mare
Insertion sort	0.08691s	Prea mare	Prea mare	Prea mare
Selection sort	Prea mare	Prea mare	Prea mare	Prea mare

Tabela 2: Timpul de rulare pentru 10^3, 10^4, 10^5 și 10^7.

5 Comparație cu literatura

Rezultatele obținute sunt în concordanță cu concluziile din literatura de specialitate. Nu s-au evidențiat discrepanțe semnificative în raport cu majoritatea studiilor care au evaluat experimental cei cinci algoritmi de sortare menționați în lucrare.

Mai jos sunt adăugate câteva referințe către anumite lucrări care aprobă afirmația de mai sus:

- `https://www.researchgate.net/publication/315662067_Sorting_Algorithms_-_A_Comparative_Study`
- `http://www.iiitdm.ac.in/old/Faculty_Teaching/Sadagopan/pdf/DAA/SortingAlgorithms.pdf`

- https://www.researchgate.net/publication/288825600_A_Comparative_Study_of_Sorting_Algorithms

6 Concluzii și direcții viitoare

Așa cum am observat, algoritmii ineficienți devin practic inutilizabili pe măsură ce volumul de date crește. În cazul de față, am oprit testarea performanței acestora pentru 10^7, deoarece nu există niciun motiv pentru care ar putea fi folosiți în practică. În schimb, algoritmii eficienți rămân extrem de utili și eficienți chiar și în fața unor volume mari de date, făcându-i esențiali pentru o gamă variată de aplicații practice.

Din analiza testelor, au fost deduse mai multe observații referitoare la performanța algoritmilor de sortare. Insertion Sort realizează cel mai scurt timp de execuție pentru elementele deja sortate, comparativ cu ceilalți algoritmi. Pentru alte tipuri de date, performanța sa este similară cu cea a Selection și Bubble Sort. Bubble Sort are timpii de execuție cei mai mari, fiind cel mai lent algoritm dintre toate cele analizate. Merge Sort și Quick Sort se dovedesc a fi cei mai eficienți algoritmi atunci când datele de intrare sunt sortate descrescător sau sunt o combinație de elemente parțial sortate și parțial aleatorii. Potrivit datelor rulate, Selection Sort afișează timpi de execuție relativ constanți pentru orice tip de date, cu diferențe minore de milisecunde față de Insertion și Bubble Sort. Merge Sort menține timpi de execuție aproape identici pentru diferitele seturi de date, în timp ce Quick Sort evidențiază o viteză remarcabilă pentru elemente aleatorii, cu diferențe notabile față de alte situații. În cazul Quick Sort, datele complet sortate sau parțial sortate generează cei mai mari timpi de execuție, dar pentru datele aleatorii, acesta este cel mai rapid algoritm dintre toate cele evaluate.

Privind spre direcțiile viitoare, există numeroși algoritmi de sortare în informatică care merită explorați experimental cât și în utilizare. Pentru cei interesați, recomand să se documenteze despre diversele opțiuni disponibile în domeniul algoritmilor de sortare.

Bibliografie

[Fri56] E. Friend. Sorting on electronic computer systems. *J. ACM*, 3:134–168, 1956.

[Hoa61] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321, July 1961.

[Ive62] Kenneth E. Iverson. *A Programming Language*. John Wiley; Sons, Inc., USA, 1962.

[Joh60] P. D. Johnson. Programming business computers. by d. d. mccracken, h. weiss and t-h. lee. [pp. 510. london: John wiley and sons, ltd., 1959. 82s.]. *Journal of the Institute of Actuaries*, 86(3):333–334, 1960.

[Knu98] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., USA, 1998.

[KT97] Jyrki Katajainen and Jesper Larsson Träff. A meticulous analysis of mergesort programs. In *Algorithms and Complexity*, pages 217–228, Berlin, Heidelberg, 1997.