Tema 1 de laborator

Aioanei Florin Verzotti Matteo Voaides Robert

Compararea unor algoritmi de sortare prin comportamentul lor asupra mai multor suite de teste

Cuprins

1 Introducere **2**

2 Algoritmi analizați **2**

3 Complexitate teoretică **2**

4 Implementare **3**

5 Metodologie experimentală **3**

6 Rezultate experimentale **4**
6.1 Numere putine, valori dense 4
6.2 Numere putine, valori dispersate 4
6.3 Numere multe, valori dense 5
6.4 Numere multe, valori dispersate 5
6.5 Numere mai multe, valori dense 6
6.6 Numere mai multe, valori dispersate 6
6.7 Numere mai multe, o singura valoare 7
6.8 Numere mai multe, 2 valori 7
6.9 Numere mai multe, 11 valori 8

7 Concluzii **8**

1 Introducere

În această lucrare ne propunem să comparăm performanțele mai multor algoritmi de sortare din punct de vedere al complexității teoretice și al performanței practice. Scopul este de a evidenția avantajele și dezavantajele fiecărui algoritm în funcție de dimensiunea și natura datelor de intrare.

2 Algoritmi analizați

- Quick Sort
  - Random Pivot
  - Median Pivot
  - Half Pivot
  - Ternary Quick Sort
- Radix Sort
  - base 10
  - base 16
  - base 2^{16}
- Merge Sort
- Intro Sort
- Tim Sort
- Shell Sort

3 Complexitate teoretică

Tabela 1: Tabel comparativ al complexităților algoritmilor analizați

Algoritm	**Best case**	**Average**	**Worst Case**
Quick Sort (Random Pivot)	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$
Quick Sort (Median Pivot)	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
Quick Sort (Half Pivot)	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$
Ternary Quick Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^2)$
Radix Sort (base 10)	$\mathcal{O}(nk)$	$\mathcal{O}(nk)$	$\mathcal{O}(nk)$
Radix Sort (base 16)	$\mathcal{O}(nk)$	$\mathcal{O}(nk)$	$\mathcal{O}(nk)$
Radix Sort (2^{16})	$\mathcal{O}(nk)$	$\mathcal{O}(nk)$	$\mathcal{O}(nk)$
Merge Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
Intro Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
Tim Sort	$\mathcal{O}(n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$
Shell Sort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n^{5/4})$	$\mathcal{O}(n^2)$

4 Implementare

Toți algoritmii au fost organizați într-un repository GitHub, structurat clar. Fiecare algoritm de sortare are propriul său header dedicat în folderul include/, iar implementarea este realizată în fișiere .cpp separate.

5 Metodologie experimentală

Pentru testarea performanțelor, am creat un sistem automatizat de rulare și înregistrare a testelor, organizat în jurul fișierului principal `main.cpp`. Setările fiecărui test sunt definite într-un fișier de configurare `test_config.csv`, unde fiecare linie specifică: `nameoftest, numberofvalues, maxvalue`. Aceste configurații sunt parcurse și procesate cu ajutorul funcției `readTestConfigs()`, iar pentru fiecare algoritm de sortare implementat în sistem, se generează un vector aleatoriu corespunzător parametrilor testului.

Toți algoritmii sunt înregistrați într-un map, ce permite selectarea și rularea lor pe baza numelui.

Timpul de execuție pentru fiecare sortare este măsurat cu funcția `measureTime()`, iar rezultatele sunt salvate în fișierul `results.csv`, folosind funcția `logResult()` incluzând: numele testului, algoritmul, dimensiunea vectorului, valoarea maximă, timpul de execuție (în secunde) și dacă vectorul rezultat este sortat corect.

Pentru o testare completă, aplicația poate fi rulată cu comanda `./main all`, caz în care se rulează toți algoritmii în toate testele definite.

Fiecare algoritm este rulat o singură dată per configurație, iar logica poate fi extinsă ușor pentru repetare multiplă și calculul unei medii. Măsurarea timpului s-a realizat utilizând funcționalități din biblioteca standard `<chrono>`.

6 Rezultate experimentale

6.1 Numere putine, valori dense

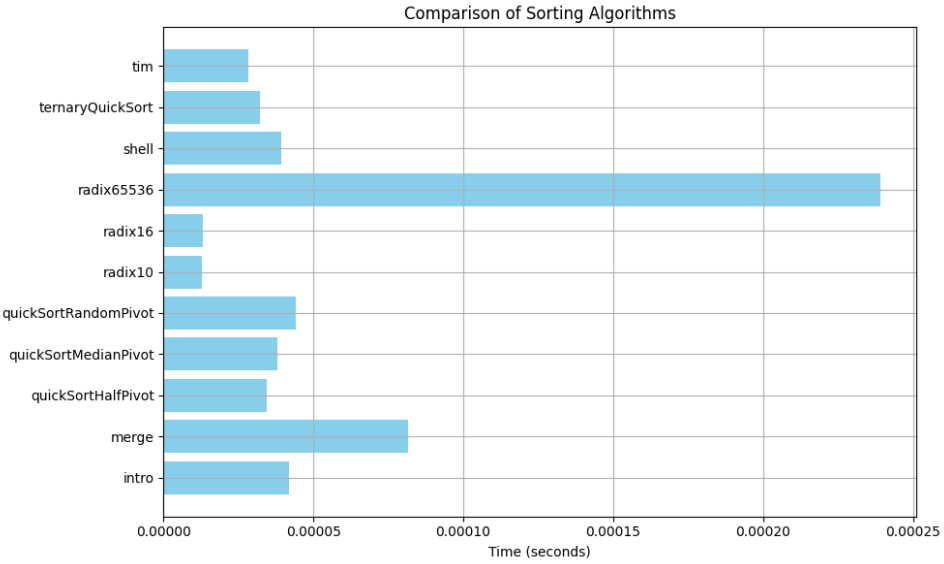

Figura 1: Pentru 1000 de numere si valori intre 0 si 10^3

6.2 Numere putine, valori dispersate

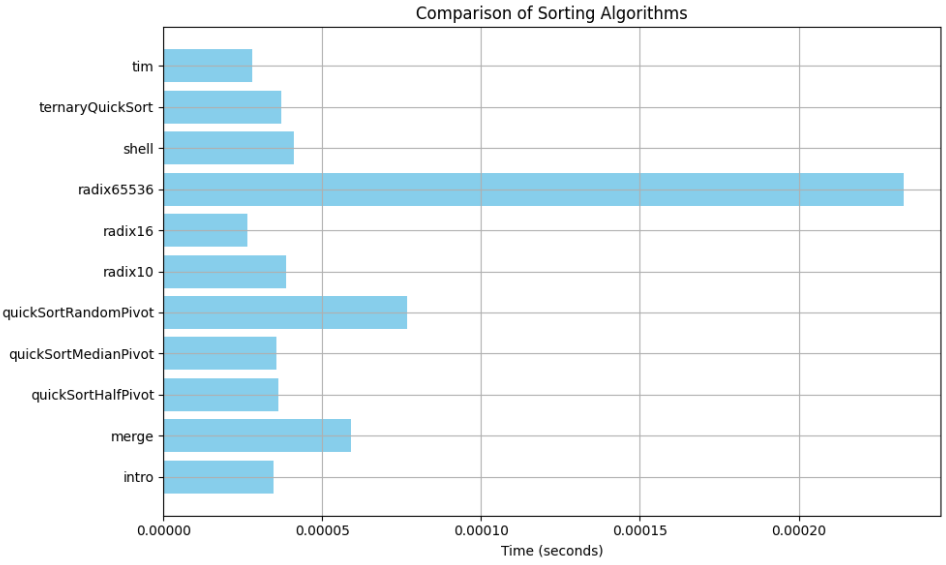

Figura 2: Pentru 1000 de numere si valori intre 0 si 10^9

6.3 Numere multe, valori dense

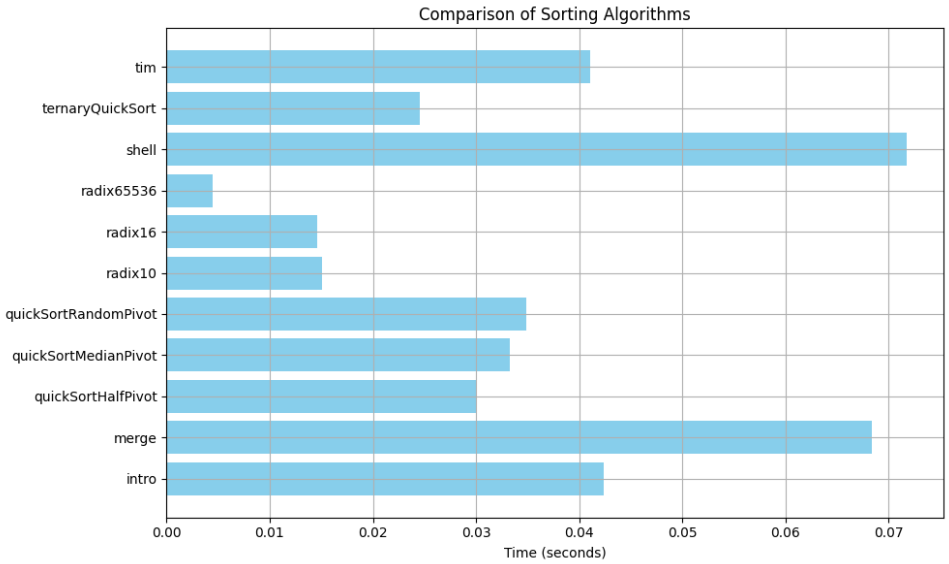

Figura 3: Pentru 10^6 numere si valori intre 0 si 10^3

6.4 Numere multe, valori dispersate

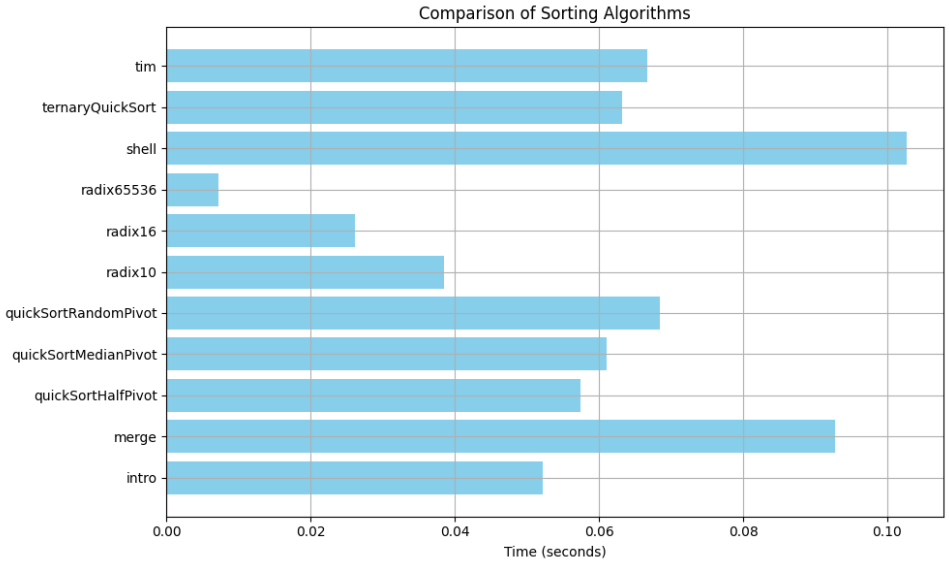

Figura 4: Pentru 10^6 numere si valori intre 0 si 10^9

6.5 Numere mai multe, valori dense

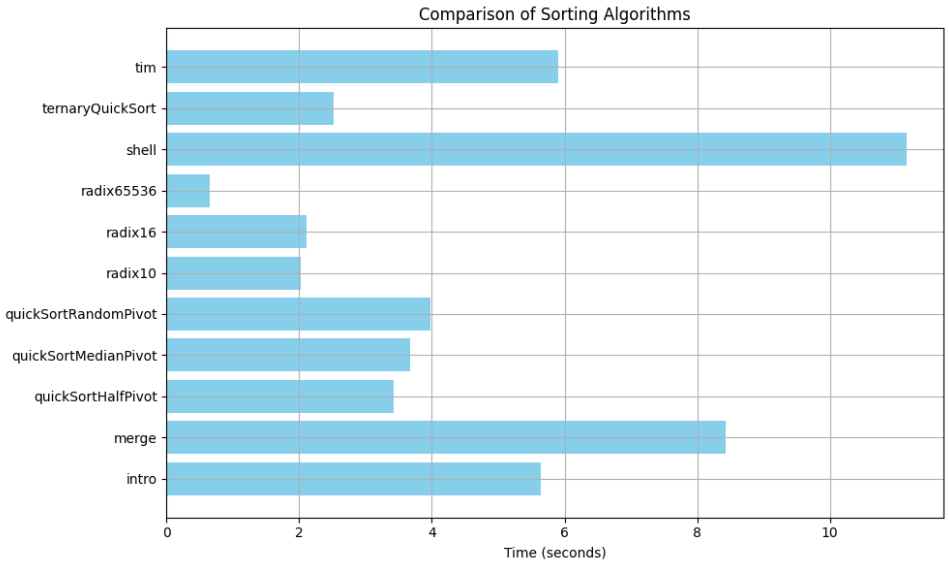

Figura 5: Pentru 10^8 numere si valori intre 0 si 10^3

6.6 Numere mai multe, valori dispersate

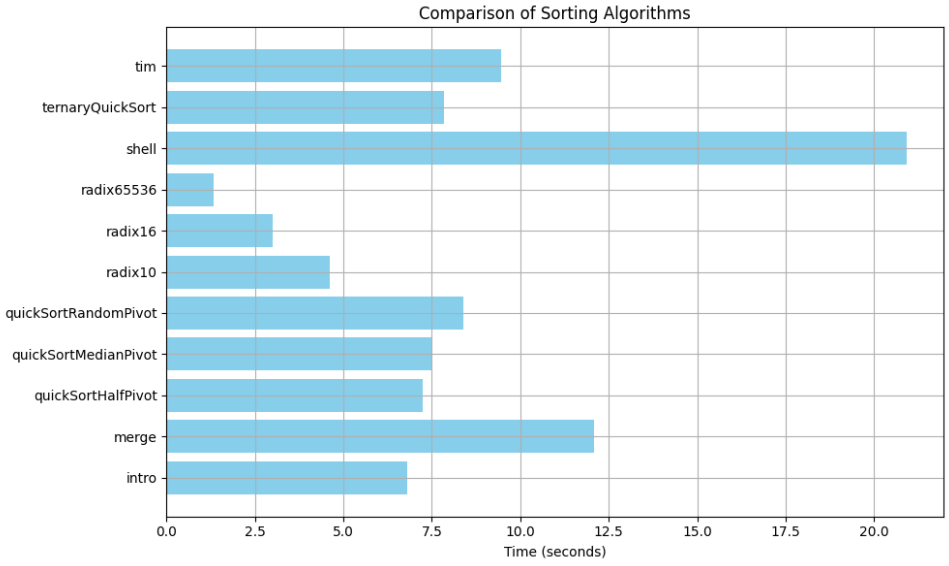

Figura 6: Pentru 10^8 numere si valori intre 0 si 10^9

6.7 Numere mai multe, o singura valoare

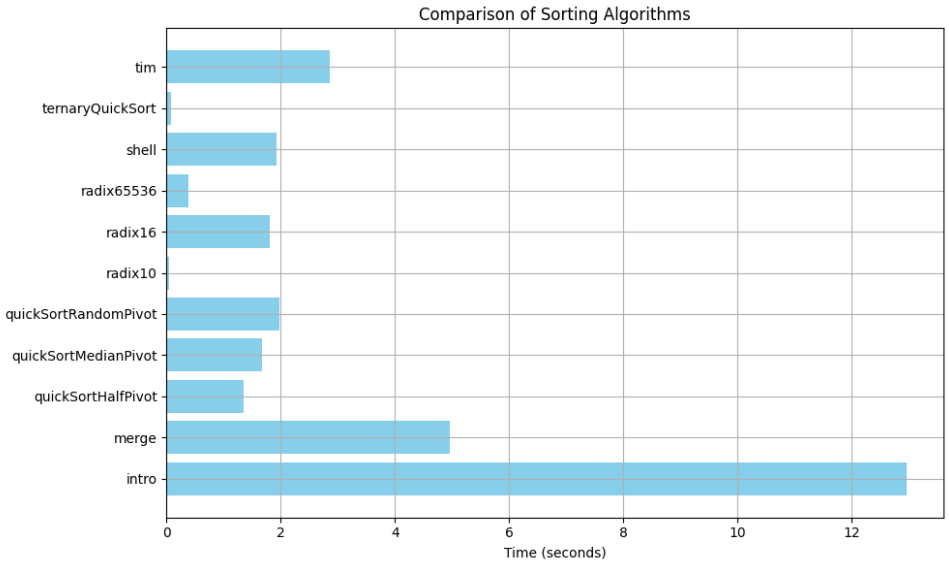

Figura 7: Pentru 10^8 numere egale cu 0

6.8 Numere mai multe, 2 valori

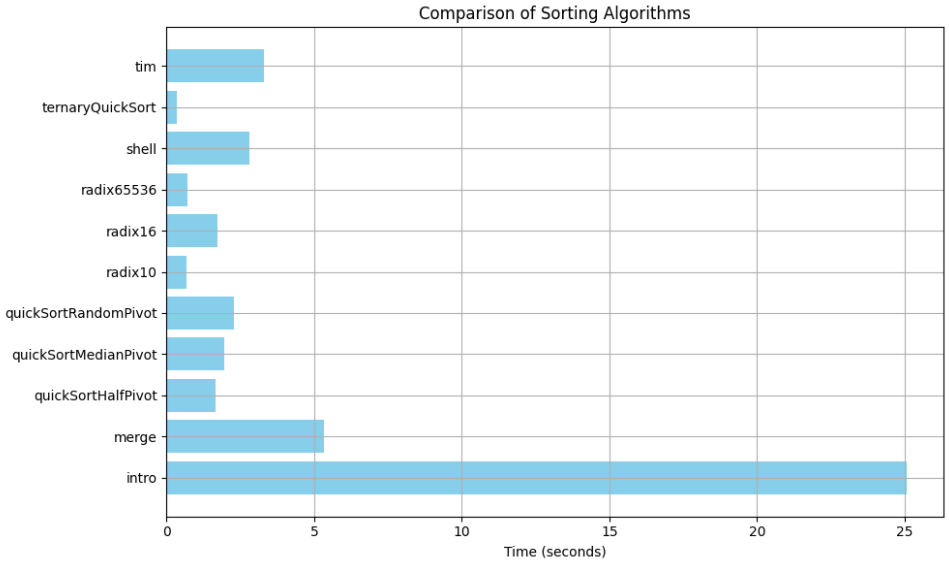

Figura 8: Pentru 10^8 numere egale cu 0 sau 1

6.9 Numere mai multe, 11 valori

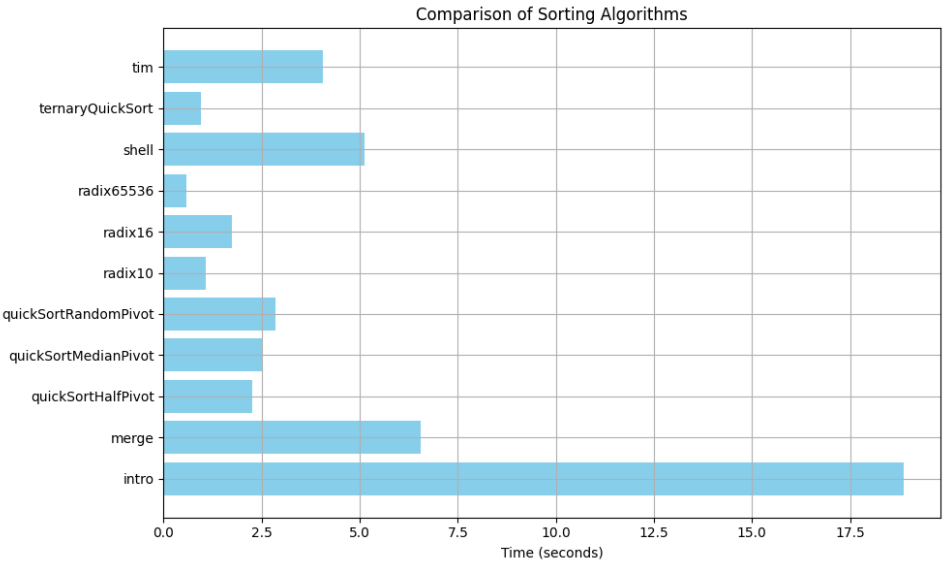

Figura 9: Pentru 10^8 numere egale cu numere de la 0 la 10

7 Concluzii

Analiza comparativă a evidențiat diferențe semnificative între algoritmii de sortare, atât în ceea ce privește complexitatea teoretică, cât și comportamentul practic în diverse scenarii. Quick Sort, în variantele sale, a demonstrat performanțe bune în medie, însă sensibilitatea la alegerea pivotului poate conduce la cazuri nefavorabile, în special pentru date deja parțial sortate sau cu distribuții neuniforme.

Radix Sort s-a remarcat ca fiind extrem de eficient pentru date numerice cu valori mici și dense, mai ales în variantele cu baze mari, dar depinde de specificul datelor și nu este o soluție universală. Merge Sort și Tim Sort s-au dovedit a fi opțiuni robuste și stabile, potrivite pentru sortări sigure și consistente indiferent de structura datelor. Tim Sort, în particular, a excelat în cazurile în care datele conțineau subsecvențe deja sortate, beneficiind de optimizările sale adaptative.

Shell Sort a avut performanțe intermediare, dar inconsistente, fiind mai puțin recomandat pentru seturi de date mari. În schimb, algoritmi hibrizi precum Intro Sort combină eficiența și siguranța, fiind potriviți în aplicații generale unde este nevoie de control asupra timpului de execuție și de evitare a celor mai nefavorabile cazuri.

În concluzie, alegerea algoritmului de sortare trebuie realizată în funcție de natura și dimensiunea datelor de intrare, precum și de cerințele specifice ale aplicației: timp de execuție, stabilitate, sau complexitate în implementare.