O comparație teoretică și experimentală a metodelor de sortare

Cosmin - Ionuț Trifu[1]

[1]Departamentul de Informatică, Facultatea de Matematică și Informatică, Universitatea de Vest Timișoara, România ,
Email: cosmin.trifu04@e-uvt.ro

May 2023

Rezumat

Sortarea reprezintă un element de bază în cadrul informaticii, iar sortarea cât mai rapidă și eficientă este un lucru bine căutat în special când vine vorba de cantități mari de date si informații, din această cauză există o gamă largă de algoritmi de sortare și mult mai multe moduri de a implementa algoritmii respectivi, fie complexi, fie foarte simpli. În această lucrare vom discuta despre și vom compara câțiva algoritmi de sortare destul de cunoscuți, atât în teorie cat și în practică. Urmărim obținerea unor date empirice din experimentele practice, compararea acestor date și într-un final, găsirea a câte unui uz pentru fiecare algoritm in parte.

Cuprins

1 Introducere

În vasta lume a informaticii, sortarea este un lucru de bază, necesar pentru o gamă vastă de aplicații care administrează o cantitate cel puțin decentă de informații/date. Sortarea, ca definiție simplă este rearanjarea unor elemente in funcție de o anumită cheie (un anumit criteriu de sortare) fie el în ordine crescătoare sau lexicografică, descrescătoare sau invers lexicografică sau în funcție de orice alt criteriu.

1.1 Scopul lucrării

Scopul acestei lucrări este de a pune în evidență câtiva algoritmi de sortare bine cunoscuți atât din punct de vedere teoretic cât și din punct de vedere practic, aceștia vor fi atât definiți cât și explicați, iar eficiența acestor algoritmi va fi pusă în discuție, cu scopul de a confirma dacă așteptările teoretice coincid cu datele experimentale obținute, se vor compara acești algoritmi cu intenția de a trage niste concluzii semnificative și de a atribui câte o folosință pentru fiecare algoritm în parte.

2 Proprietăți ale unui algoritm de sortare

Un algoritm de sortare este caracterizat de mai multe proprietati pe care le prezintă, cateva din acestea sunt prezentate mai jos:

2.1 Complexitate temporală

Cea mai importantă proprietate care ne ajută să clasificăm algoritmii de sortare este complexitatea temporală. Aceată proprietate presupune performanta fiecărui algoritm din punct de vedere a duratei timpului de executie, în care există atât cazuri favorabile, cazuri nefavorabile cât și cazuri aleatorii.

Complexitatea temporală a unui algoritm este reprezentată de o functie, iar ca notație avem "The Big-O" (mărginirea superioară).[?]

De exemplu, in cel mai rau caz, Insertion Sort are o complexitate de $O(n^2)$ si in cel mai bun caz de $O(n)$.

2.2 Complexitate spatială

Această proprietate este una semnificativă in alegerea tipului de algoritm de sortare pe care dorim să îl folosim, reprezentând spațiul total folosit de un algoritm în funcție de dimensiunea datelor de intrare si spatiul auxiliar folosit de algoritm, de asemena se utilizeaza notația "Big-O" pentru exprimarea complexității din punct de vedere a spațiului.

Un algoritm de sortare pe loc este acela care nu necesită spațiu auxiliar pentru a sorta elementele(folosirea spațiului auxiliar pentru apeluri recursive nu schimbă această proprietate).

2.3 Stabilitate

Un algoritm de sortare stabil păstrează ordinea relativă a elementelor egale.

3 Descrierea algoritmilor de sortare

In această parte a lucrării vom face analiza separată a fiecărui algoritm de sortare în parte: fiecare algoritm va fi definit, se vor discuta proprietățile lor și se vor lista atât avantajele cât si dezavantajele metodelor de sortare. Algoritmii propuși spre analiză se încadrează in doua mari categorii, algoritmi care folosesc comparații si algoritmii care nu folosesc comparații, astfel algoritmii propuși se împart astfel:

Algoritmi bazați pe comparații:

1. Bubble Sort
2. Selection Sort
3. Insertion Sort

4. Merge Sort

5. Quick Sort

Algoritmi fără comparații:

1. Counting Sort

2. Radix Sort

3.1 Bubble Sort

Bubble Sort[?][?] este unul dintre cei mai simpli algoritmi de sortare. Algoritmul compară fiecare element din tablou cu elementele vecine și le interschimbă dacă nu sunt în ordinea potrivită, această etapă se repetă până cand tabloul este sortat. De aceea sortarea devine din ce în ce mai înceată cu cât se adaugă mai multe elemente in tablou.

3.1.1 Analiza eficienței

Bubble Sort este considerat a fi cea mai ineficientă metodă de sortare pentru că complexitatea temporală in cazul mediu și în cel mai rău caz este de $O(n^2)$, unde n reprezintă numărul de elemente care trebuie sortate.

3.2 Selection Sort

Selection Sort[?][?] este un algoritm de sortare simplu, mai eficient decât Bubble Sort, și functionaza folosind comparații pentru a găsi cel mai mare element, îl inlocuieste cu valoarea din capatul tabloului, si repetă cautarea celei mai mari valori următoare, formând un tablou sortat în dreapta si unul nesortat in stânga, până când tot tabloul este sortat.

3.2.1 Analiza eficienței

Dat fiind faptul că Selection Sort este un algoritm similar cu Bubble Sort in sensul că este iterativ, pentru fiecare n elemente dintr-un

tablou este nevoie de n-1 iterații, așadar complexitatea temporală a acestui algoritm ajunge la $O(n^2)$ insă este mai eficient decat Bubble Sort.

3.3 Insertion Sort

Insertion Sort[?][?] este un algoritm relativ eficient și simplu pentru tablouri mici sau sortate. Metoda aceasta de sortare inserează fiecare element din tabloul nesortat la poziția potrivită din partea tabloului care este sortată (care la inceput este de dimensiune 1), această acțiune este repetată până când tabloul este sortat integral.

3.3.1 Analiza eficienței

Insertion Sort este de departe cel mai bun algoritm dintre algoritmii tradiționali prezentați în această lucrare, în ciuda complexității $O(n^2)$ în cazul unui tablou nesortat, este mai rapid decât Bubble Sort și Selection Sort, de asemenea are și un caz favorabil de $O(n)$ atunci când tablourile sunt sortate. Din punct de vedere al spațiului auxiliar, Insertion Sort necesită $O(1)$.

3.4 Merge Sort

Merge Sort[?] este un algoritm foarte eficient de tip divide and conquer. Această metodă de sortare functionează prin impărțirea tabloului nesortat în două intr-un mod recursiv până când se ajunge la tablouri de mărime indivizibilă (1 element). Prin definitie un tablou de lungime 1 este considerat sortat. După etapa de divizare, fiecare subtablou este interclasat rezultând un tablou integral sortat.

3.4.1 Analiza eficienței

Merge Sort are o complexitate in caz nefavorabil și mediu de $O(n \log n)$, însă din punct de vedere al spațiului auxiliar Merge Sort este defavorizat, necesitând O(n) spatiu auxiliar pentru n elemente, facându-l ineficient pentru aplicații cu memorie redusă.

3.5 Quick Sort

Quick Sort[?][?], la fel ca Merge Sort, se folosește de paradigma divide and conquer pentru a rezolva problema sortării. Algoritmul selectează cate un pivot din partea nesortată a tabloului și împarte tabloul în două părți, primul subtablou fiind format din elementele mai mici decât pivotul iar a doua din cele mai mari decât pivotul(o partiționare a tabloului). Pentru ambele subtablouri, această operațiune este repetată recursiv până când tabloul este sortat.

3.5.1 Analiza eficienței

Avantajul cel mai mare al Quick Sort-ului este eficiența temporală, fiind in medie mai rapid chiar și decât Merge Sort, având o complexitate temporală similară cu Merge Sort de $O(n \log n)$, un mare dezavantaj este reprezentat de prezența cazului nefavorabil în care tabloul este deja sortat, ajungându-se la $O(n^2)$, din punct de vedere al complexității spațiale, algoritmul nu necesită spațiu auxiliar pentru sortarea în sine, ci numai pentru apelurile recursive, adică $O(n)$.

3.6 Counting Sort

Counting Sort[?][?] este un algoritm care nu folosește comparații pentru a rezolva problema sortării, functionează prin contorizarea cheilor de sortare distincte si determinarea pozițiilor lor, apoi aplicarea unei sume prefixate acelor contoare.

3.6.1 Analiza eficienței

Complexitatea temporală acestui algoritm este de $O(n + k)$ unde k reprezintă valoarea maximă a elementelor din tabloul de lungime n. Counting Sort necesită un spațiu auxiliar de $O(k)$.

3.7 Radix Sort

Radix Sort[?] este un algoritm similar cu Counting Sort (Counting Sort fiind adeseori intâlnit ca subrutină a acestui algoritm) care folosește chei de sortare(cifre). El sortează fiecare cifră a fiecărui

număr din datele de intrare de la cea mai nesemnificativă până la cea mai semnificativă.

3.7.1 Analiza eficienței

La fel ca și la Counting Sort, complexitatea temporală depinde de intervalul de valori al tabloului, având $O(n*k)$ unde k este valoarea maximă din tabloul de lungime n. Iar spațiul auxiliar necesar este de $O(n+k)$.

4 Implementare

Structura programului de testare prezintă atât funcțiile de măsurare a timpului de executie cât si multitudinea de algoritmi de sortare pentru care se fac aceste teste. Programul este capabil de a genera un număr mare de valori aleatorii, pentru a avea teste cât mai concludente. Dimensiunea tablourilor a variat de la 100 de elemente pana la 1.000.000. Pentru generarea elementelor aleatorii s-a folosit funcția rand() si srand() pentru seed-ul randomizării. Toate măsurătorile făcute au fost cu ajutorul bibliotecii time.h, mai exact funcția clock(), după care au fost scrise într-un fișier de ieșire.

5 Comparația algoritmilor

Metodă de sortare	Caz nefavorabil	Caz favorabil	Caz mediu	Stabilitate
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Da
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Nu
Insertion Sort	$O(n^2)$	$O(n)$	$O(n^2)$	Da
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Da
Quick Sort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$	Nu

Tabela 1: Compararea algoritmilor bazați pe comparare

În secțiunea 3 s-a parcurs lista algoritmilor de sortare propuși pentru a reaminti atât modul de funcționare în linii mari cât și câteva aspecte ale complexității pentru fiecare algoritm in parte dar și

niște factori pro si contra utilizării algoritmilor. În aceasta sectiune se va face atât comparația teoretică cât și cea experimentală atât folosind tabele cât și grafice.

În tabelul 1 se pot observa diferențe MAJORE între algoritmii tradiționali (Bubble, Selection, Insertion) și cei eficienți (Merge,Quick).

Metodă de sortare	Caz nefavorabil	Caz favorabil	Caz mediu	Stabilitate
Counting Sort	$O(n+k)$	$O(n+k)$	$O(n+k)$	Da
Radix Sort	$O(n \cdot k)$	$O(n \cdot k)$	$O(n \cdot k)$	Da

Tabela 2: Compararea algoritmilor care nu folosesc comparări

În tabelul 2 se poate observa faptul că acești algoritmi au complexități temorale liniare însă depind în totalitate de intervalul valorilor elementelor.

5.1 Comparația experimentală

Testele practice se vor face atât pe tablouri generate aleatoriu cât și pe tablouri sortate pentru analiza comportamentul metodelor de sortare. Acest comportament va fi ilustrat atât pe grafice logaritmice cât și pe tabele de valori.

5.1.1 Testul 1

Testul 1 a fost făcut asupra unor tablouri cu valori cuprinse între 0 și 50000, generate aleatoriu cu ajutorul rand(), mai jos se poate vedea comportamentul metodelor de sortare care folosesc comparații.

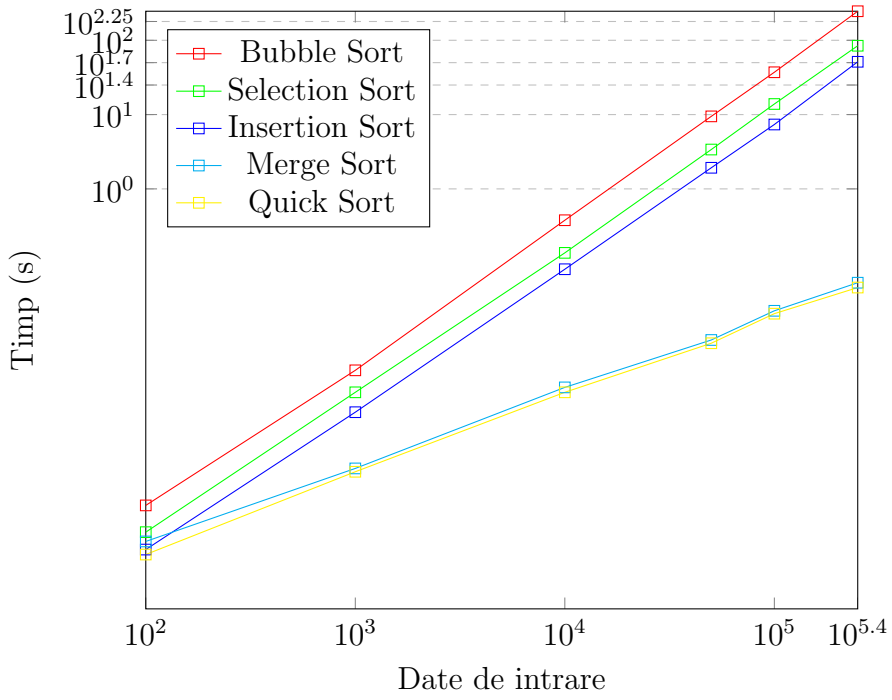

Figura 1: Comportamentul metodelor bazate pe comparații (grafic logaritmic).

Din Figura 1 se pot observa două tipuri de creșteri ale timpului în secunde în funcție de timp, reprezentând cele doua complexități temporale încadrate de acest grafic: $O(n^2)$ și $O(n \log n)$. Există o diferență clară între metodele pătratice și cele logaritmice, algoritmii de sortare pătratici având o rată de creștere mult mai dramatică față de algoritmii eficienți ca Merge Sort si Quick Sort.

Din grafic se poate observa că Quick Sort, în cazul tablourilor random, este cea mai rapidă dintre metodele abordate în această lucrare, iar Bubble Sort cea mai înceată. Dintre algoritmii cu complexitate pătratică Insertion Sort depășeste la performantă Bubble Sort și Selection Sort.

Mai jos este tabelul cu valori pe care este bazată Figura 1.

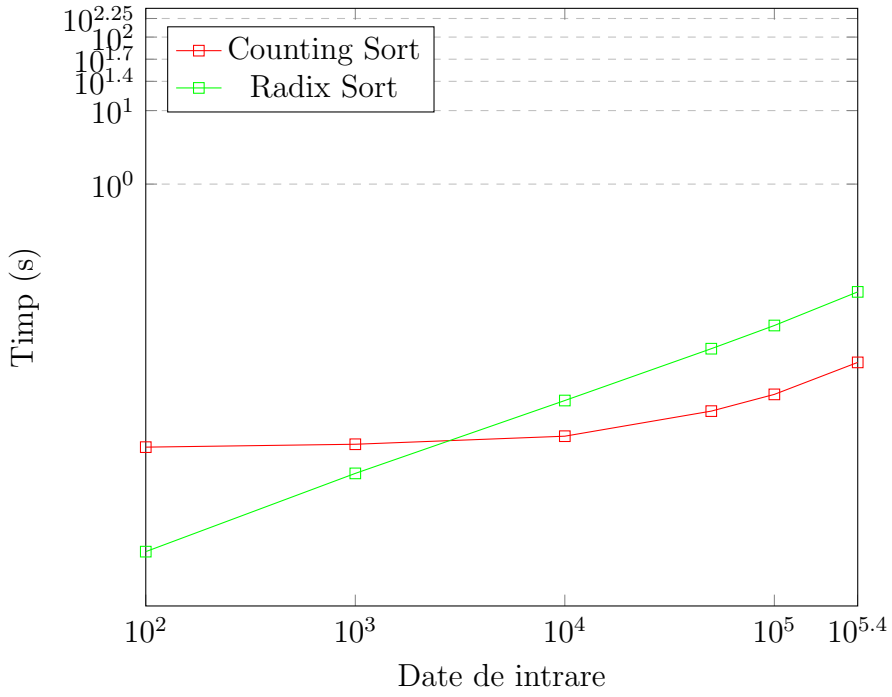

Figura 2: Comportamentul metodelor care nu sunt bazate pe comparații (grafic logaritmic).

Deoarece ar fi greșit să comparăm algoritmii tradiționali cu cei care nu sunt bazați pe comparații, Counting Sort si Radix Sort au fost luați separat. Mai jos este Figura 2, care conține graficul logaritmic a acestor metode de sortare. Pentru testele pe acești algoritmi, vom lua domeniul de valori [0, 50000].

Sortare	100	1000	10000	50000	100000	250000
Bubble	0.000055	0.003617	0.378660	9.462800	37.131100	245.975000
Selection	0.000024	0.001830	0.137730	3.385700	13.882200	84.421000
Insertion	0.000014	0.000988	0.082940	1.923900	7.355800	51.354000
Merge	0.000018	0.000173	0.002131	0.009259	0.022875	0.054600
Quick	0.000012	0.000156	0.001830	0.008373	0.020926	0.047006

Tabela 3: Datele testului 1 pentru algoritmii tradiționali

Din Figura 2 remarcăm că Radix Sort este inițial mai rapid decât Counting Sort (pentru tablouri mici), urmând ca acest lucru să se schimbe începând cu tablourile de 10000 de elemente.

Sortare	100	1000	10000	50000	100000	250000
Counting	0.000264	0.000289	0.000372	0.000816	0.001378	0.003754
Radix	0.000010	0.000116	0.001137	0.005764	0.011920	0.034180

Tabela 4: Datele testului 1 pentru algoritmii netradiționali

Mai jos se va avea la dispoziție tabelul de valori pe care este bazat Figura 2, pentru clarificarea oricărei nelămuriri.

5.1.2 Testul 2

Pentru testul 2 vom considera tablouri cu valori cuprinse între 0 și 50000 sortate, pentru a analiza comportamentul metodelor de sortare (în care pentru unele este caz favorabil, pentru altele nefavorabil, iar pentru altele nu conteaza).

Mai jos avem Figura 3, care reprezinta rezultatele din testul 2 pentru algoritmii tradiționali de sortare.

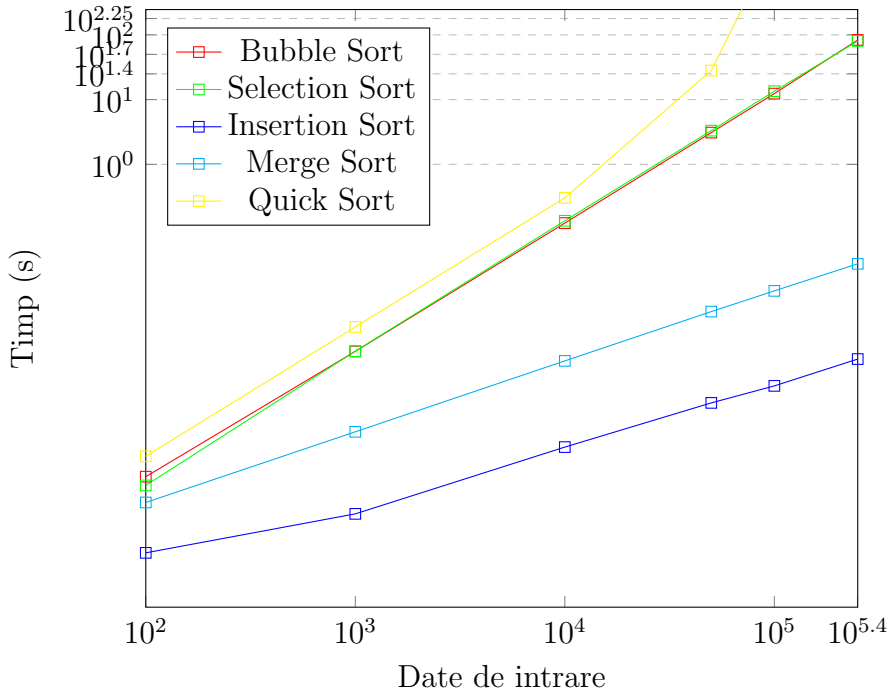

Figura 3: Comportamentul metodelor bazate pe comparații atunci când tablourile sunt deja sortate (grafic logaritmic).

În figura 3 se văd câteva schimbări majore în performanța algoritmilor de sortare; atunci când tablourile sunt sortate algoritmului Quick Sort îi scade performanta DRAMATIC, ajungând sa fie cel mai incet dintre algoritmii tradiționali și situația inrăutațindu-se major cu cât se adauga mai multe elemente în tablou.

Mai jos avem tabelul cu datele reieșite din testul 2 pentru algoritmii tradiționali.

Pentru incheierea testului 2 mai rămân de observat schimbările (dacă există) în comportamentul algoritmilor netradiționali, prezente jos, în Figura 4.

În figura 4 nu se poate observa nici o schimbare relativă intre cele 2 metode, oferită de caracterul mediu al complexității algoritmilor

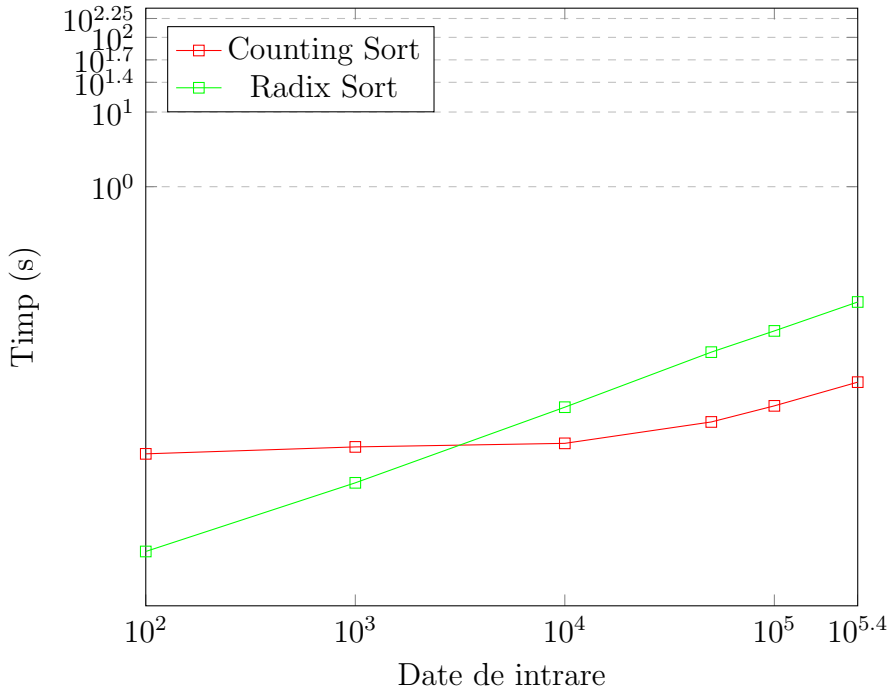

Figura 4: Comportamentul metodelor care nu sunt bazate pe comparații atunci când tablourile sunt sortate (grafic logaritmic).

Sortare	100	1000	10000	50000	100000	250000
Bubble	0.000015	0.001293	0.123890	3.094700	12.379000	83.496000
Selection	0.000011	0.001302	0.134480	3.318500	13.476000	80.504520
Insertion	0.000001	0.000004	0.000043	0.000207	0.000379	0.000984
Merge	0.000006	0.000074	0.000922	0.005324	0.011120	0.029100
Quick	0.000031	0.003082	0.303520	28.125543	2541.127000	mult

Tabela 5: Datele testului 2 pentru algoritmii tradiționali

Sortare	100	1000	10000	50000	100000	250000
Counting	0.000264	0.000327	0.000365	0.000706	0.001160	0.002410
Radix	0.000013	0.000108	0.001113	0.006080	0.011700	0.028560

Tabela 6: Datele testului 2 pentru algoritmii netradiționali

(nu există cazuri favorabile/nefavorabile).

6 Concluzii și direcții viitoare

Acest scurt articol a analizat și comparat 7 dintre cei mai cunoscuți algoritmi de sortare atat din punct de vedere teoretic cât și practic. Din acest studiu reiese că nu numai complexitatea teoretică, de pe foaie, contează în eficiența unui algoritm, ci și datele de intrare, valorile datelor de intrare și cantitatea acestor date.

Bibliografie

[1] Shailendra Mishra Ashutosh Bharadwaj. Comparison of sorting algorithms based on input sequences.

[2] C.A.R. Hoare. The computer journal. 5:10–16, 1962.

[3] Ashok Kumar Karunanithi. A survey, discussion and comparison of sorting algorithms.

[4] Abdallah Mahmoud Ibrahim AlTurani Nabeel Imhammed Zanoon Khalid Suleiman Al-Kharabsheh, Ibrahim Mahmoud AlTurani. Review on sorting algorithms a comparative study.

[5] Donald Ervin Knuth. The Art of Computer Programming. Addison Wesley, second edition, 1998.

[6] Kevin Williams David Gregg Paul Biggar, Nicholas Nash. An experimental study of sorting and branch prediction.

[7] Harold H. Seward. Information sorting in the application of electronic digital computers to business operations.

[8] Thomas Cormen. Charles Leiserson. Ronald Rivest. Clifford Stein. Introduction to algorithms. fourth edition.

Cod GitHub: https://github.com/Trifu-Cosmin/MPI

Implementări algoritmi: https://pastebin.com/1D81mkkf