O comparaţie teoretică şi experimentală a metodelor de sortare

Robert Lissi
Student profil Informatica,
Universitatea de Vest, Timişoara
Email: robert.lissi01@e-uvt.ro

Iunie 2021

Cuprins

1 Introducere

Sortarea este un proces de rearanjare a unei liste de elemente în ordinea corectă, deoarece mutarea elementelor într-o anumită ordine este mai eficientă decât mutarea unor elemente aleatoriu .Sortarea și căutarea sunt printre cele mai comune procese de programare,de exemplu luam baza de date de aplicații dacă dorim să păstram informațiile și să le regăsim cu ușurință trebuie să păstram informațiile într-o ordine, de exemplu, ordinea alfabetică, ordinea crescătoare/descrescatoare, ordinea în funcție de nume, id-uri, ani, departamente, etc.Numărul mare de algoritmi dezvoltat pentru a îmbunătățirea sortari, cum ar fi urmatoarele metode de sortarea:Merge sort,Bubble sort, Insertion Sort, Quick Sort, etc, fiecare dintre acestea are un mecanism diferit de reordonare a elementelor care mărește performanța și eficiența aplicațiilor practice și reduce complexitatea în timp a fiecăruia. Atunci când se face o comparație între diferiți algoritmi de sortare, există mai mulți factori care trebuie luați în considerare.
Primul dintre ei este complexitatea în timp, complexitatea în timp a unui algoritm determina timpul necesar unui algoritm pentru a fi executat. Acest factor diferă de la un algoritm de sortare la altul în funcție de mărimea datelor pe care dorim să le ordonăm. Al doilea factor este stabilitatea , algoritmul păstrează elementele cu valori egale în aceeași ordine relativă la ieșire, așa cum erau la intrare. Unii algoritmi de sortare sunt stabili prin natura lor, cum ar fi Insertion Sort, Merge Sort, Bubble Sort, în timp ce unii algoritmi de sortare nu sunt stabili, cum ar fi Quick Sort, orice algoritm de sortare dat care nu este stabil poate fi modificat pentru a fi stabil .
Al treilea factor este spațiul de memorie, algoritmii care utilizează tehnici recursive au nevoie de mai mult spațiu de memorie,copiile datelor de sortare care afectează spațiul de memorie .

1.1 Motivaţie

Sortarea este un domeniu de cercetare foarte solicitat în informatică și unul dintre cele mai importante domenii de cercetare de bază în informatică. Problema algoritmilor de sortare a atras numeroase studii în domeniul informaticii. Scopul principal al utilizării algoritmilor de sortare este de a face ca algoritmi de căutat, de inserat și de șters sa fie cat mai eficenti. Analizăm algoritmi de sortare: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort și Quick Sort, complexitatea lor în timp si spatiu. În plus, din aspectele legate de datele de intrare, s-au obținut câteva rezultate pe baza experimentelor. Astfel, am analizat că, atunci când dimensiunea datelor de intrare este mică, Sortarea prin Inserție sau Sortarea prin Selecție sunt mai eficente, iar atunci când dimensiunea secventei este ordonată,Bubble sort, acesta fiind mai bun in acest caz.
În această lucrare, prezint un rezultat general al analizei de sortare al algoritmilor de sortare și a proprietăților acestora. Problema abordată este importantă pentru că există un număr mare de aplicaţii în practică ce ar beneficia de soluţii pentru această problemă, dar soluţiile existente sunt neadecvate pentru că sunt foarte complexe din punct de vedere al timpului şi memoriei, şi totodată imple-

mentarea acestora este anevoioasă. [1]

2 Prezentarea formală a problemei şi soluţiei

2.1 Formalizarea problemei

Comparatia intre algoritmi de sortare
Toți algoritmi de sortare prezentati (Selection Sort, Insertion sort, Merge sort, Quick sort, Bubble Sort,Radix Sort,Counting Sort) au fost implementați în limbajul de programare C++ și testați pentru secvente de date de intrare de lungimi diferite. Rezultatul arată că, pentru o secventa mica de date de intrare, performanțele pentru acesti algoritmi sunt apropiate, dar pentru intrări mai lungi de date de intrare Quick Sort este cel mai rapid si algoritmul de sortarea prin selectie este cel mai lent.
Analiza complexitatii unui algoritm are ca scop estimarea volumului de resurse de calcul necesare pentru executia algoritmului.Resurse:
Spatiul de memorie necesar pentru stocarea datelor pe care le prelucreaza algoritmul.Timpul necesar pentru executia tuturor prelucrarilor specicate in algoritm. Aceasta analiza este utila pentru a stabili daca un algoritm utilizeaza un volum de resurse aceptabil pentru rezolvarea unei probleme. [5] [3]

2.2 Soluţia algoritmică

2.2.1 Insertion Sort

Sortarea prin Inserție este foarte asemănătoare cu sortarea prin Selecție. Este un algoritm de sortare simplu care construiește un vector final sortat. Complexitatea timpului de executie este O(n*n),acest algoritm este mult mai puțin eficient în cazul unor liste mari de numere,decât alti algoritmii mai avansați, cum ar fi Quick Sort,Heap sort, sau Merge sort. Cu toate acestea,sortarea prin inserție oferă mai multe avantaje. Implementare este simplă și eficenta pentru seturi de date mici. Sortarea prin inserție este un algoritm simplu de sortare care funcționează similar cu modul în care amestecam cărțile de joc . Vectorul este practic împărțit într-o parte sortată și una nesortată. Valorile părții nesortate sunt selectate și plasate în poziția corectă în partea sortată. Algoritmul:Pentru a sorta un vector de mărime n în ordine crescătoare:
1: Iteram de la v[1] la v[n] in vector.
2: Comparam elementul curent (cheia) cu predecesorul său.
3: Dacă elementul cheie este mai mic decât predecesorul său,il comparam cu elementele anterioare. Mutam elementele mai mari cu o poziție spre dreapta pentru face spațiu pentru elementul schimbat. [3] [8]

2.2.2 Quick Sort

Acest algoritm de sortare alege un element ca pivot și partiționează matricea dată în jurul pivotului ales. Există multe versiuni diferite de QuickSort care

aleg pivotul în moduri diferite.
a. Alegem întotdeauna primul element ca pivot.
b. Alegem întotdeauna ultimul element ca pivot
c. Alegem un element aleatoriu ca pivot.
d. Alegem mediana ca pivot. Procesul cheie în Quick Sort este functia partititonare().Scopul partițiilor este, având în vedere un vector și un element x din vector luat ca pivot,il punem pe x la poziția sa corectă în matrice sortată și punem toate elementele mai mici(mai mici decât x) înainte de x și punem toate elementele mai mari (mai mari decât x) după x. Toate acestea trebuie făcute în timp liniar. [2]

```
 Pseudocode pentru functia recursiva pentru algoritmul
 Quick Sort:/*stanga indice de inceput ,dreapta indice de sfarsit
QuickSort(v[],stanga,dreapta)
{ if (stanga < dreapta) {/* x este indexul de partitionare*/
        x = partitionare(v, stanga, dreapta);
        QuickSort(v, stanga, x - 1);  // inainte de x
        QuickSort(v, x + 1, dreapta); // dupa x  }}
```

[3]

2.2.3 Selection Sort

Sortarea prin Selecție este cel mai simplist algoritm, dar este destul de ineficient. În sortarea prin selecție,trebuie să găsim cel mai mic element care face scanarea liniară și să-l mutăm în față(schimbându-l cu elementul din față).Apoi, trebuie să găsim al doilea cel mai mic element și să îl mutăm de la locul său, efectuând din nou o scanare liniară. Se continuă astfel până când toate elementele sunt la locul lor. În algoritmul de sortare prin selectie, se mențin doi subvectori într-o matrice dată. Primul subvector care este deja sortat și al doilea care rămâne care nu este sortat.La fiecare iterație a sortării prin selecție, cel mai mic element din matricea secundară nesortată este ales și mutat în tabloul secundar sortat. [6]

2.2.4 Counting Sort

Sortarea prin Numărare este o tehnică de sortare bazată pe chei cuprinse într-un anumit interval. Funcționează prin numărarea numărului de elemente care au valori cheie distincte. Apoi se efectuează o operație aritmetică pentru a calcula poziția fiecărui obiect în secvența de ieșire.

2.2.5 Shell Sort

Shell Sort este un algoritm de sortare care sortează mai întâi elementele aflate la distanță mai mare unele de altele și reduce succesiv intervalul dintre elementele care urmează să fie sortate.Este o verianta extinsa a sortării prin inserție.Datorita

algoritmul Shell Sort, elementele sunt sortate la un anumit interval.Intervalul dintre elemente este redus treptat în funcție de secvența de date de intrare utilizată.

2.2.6 Bubble Sort

Bubble Sort este un algoritm de sortare care compară două elemente învecinate și le interschimbă dacă nu sunt în ordinea corecta.
1.Pornind de la primul indice, se compară primul și al doilea element.
2.Dacă primul element este mai mare decât al doilea, acestea se vor interschimba.
3.Acum, compar al doilea și al treilea element. Interschimb dacă nu sunt în ordine corecta. 4.Procesul de mai sus continuă până ajungem la ultimul element. După fiecare iterație, cel mai mare element dintre elementele nesortate este plasat la sfârșitul vectorului.La fiecare iterație, comparația are loc până la ultimul element nesortat. [4]

2.2.7 Merge Sort

Merge Sort este unul dintre cei mai populari algoritmi de sortare care se bazează pe principiul algoritmului Divide and Conquer.Problema este împărțită în mai multe subprobleme. Fiecare subproblemă este rezolvată individual. Dupa, subproblemele sunt combinate pentru a forma soluția finală.Presupunand ca avem de sortat un vector P[].O subproblema arfi ca trebuie sa sortam un subvector din acest vector de la indicele c pana la indicele f astfel P[c...f].

`Etapa de impartire(Divide Step)`

Daca este la jumatatea distantei intre c si f, atunci atunci putem imparti vectorul P[] in doi subvectori ,vectorul P[c..e] si P[e+1..f].

`Etapa de cucerire(Conquer Step)`

La etapa de cucerire(Conquer),incerc sa sortez cei doi subvectori P[c..e] si P[e+1..f]. Dacă nu am ajuns încă la cazul de bază, împărțim din nou cei doi subvectori și încerc să ii sortez din nou.

`Etapa de combinare(Combine Step)`

Cand pasul de cucerire(conquer) ajune la cazul de baza si obtinem cei doi subvectori P[c..e] si P[e+1..f] pentru a forma vectorul P[c..f],combinam cei doi subvectori pentru a obtine vectorul sortat P[c..f]. [7]

3 Implementare

3.1 Insertion Sort

```
void SortareInsertie(int v[], int n){ int i, x, j;
    for (i = 1; i < n; i++) { x = v[i]; j = i - 1;
```

```
/*Mutam elementele din v[0...i-1] care sunt mai mari decat x ,cu o
pozitie in fata fata de pozitia lor actuala.
*/ while (j >= 0 && v[j] > x)
{ v[j + 1] =v[j]; j = j - 1;} v[j + 1] = x;}}
Timpul de executie:O(n^2).
Sortarea prin inserție necesită timp maxim pentru sortare
dacă elementele sunt sortate în ordine inversă.
Și este nevoie de timp minim
când elementele sunt deja sortate.
```

Verificarea corectitudini algoritmului de Sortare prin insertie:

```
 Clasele de complexitate(eficiența):
T(n)= Omega(n) ,T(n) = O(n^2)
```

[8]

3.2 Quick Sort

Implementarea algoritmului QuickSort in C++

```
void swap(int* x, int* y) {int z = *x; *x = *y; *y = z; }
//partitionam Vectorul folosind ultimul element drept pivot.
int partitionare(int v[], int st, int dr)
{ int pivot = v[dr];//pivot-ul int i = (st - 1);
for (int j = st; j <= dr- 1; j++)
{/daca elementul curent este mai mic decat
  //pivotul,crestem elementul din stanga
  //interschimbam de pe pozitile i si j
    if (v[j] <= pivot)  {  i++;
    //crestem indicele elementului mai mic
            swap(&v[i], &v[j]); } }
            swap(&v[i + 1], &v[dr]);  return (i + 1); }
void QuickSort(int v[], int stanga, int dreapta)
{ if (stanga < dreapta) {/* partionam vectorul */
int pivot = partitionare(v,stanga,dreapta);
    //sortam fiecare subvector separat
QuickSort(v, stanga, pivot - 1); QuickSort(v, pivot + 1, dreapta);}}
```

[3]

3.3 Selection Sort

```
void swap(int *x, int *y) { int aux = *x; *x = *y; *y = aux;}
void SelectionSort(int v[], int n) {
  for (int j = 0; j < n - 1; j++) { int indice_min = j;
    for (int i = j + 1; i < n; i++)
{//Pentru a sorta descrescator vectorul,
```

```
    //trebuie sa schimbam > cu <
  //alegem elementul minim din fiecare parcurgere a for-ului
if (v[i] < v[indice_min])      indice_min = i;}
//punem minimul la pozitia corecta swap(&v[indice_min], &v[j]);}}
```

[6]

3.4 Counting Sort

```
void Numarare(char v[])
{//vectorul in care se vor regasi caracterele sortate
    char afisare[strlen(v)];
// Creați un vector de numărare pentru a stoca numărul de caractere
int nr_apariti[n + 1], i;
    memset(nr_apariti, 0, sizeof(nr_apariti));
    //Stocam numarul de apariti ale fiecarui caracter
    for (i = 0; v[i]; ++i) ++nr_apariti[v[i]];
//  nr[i] sa contina actuala pozitie a caracterului in vectorul afisare_[]
    for (i = 1; i <= n; ++i)
    nr_apariti[i] += nr_apariti[i - 1];
 //construim vectorul de afisare
 for (i = 0; v[i]; ++i) {
 afisare[nr_apariti[v[i]] - 1]= v[i]; --nr_apariti[v[i]]; }
/*for (i = sizeof(v)-1; i>=0; --i)
{afisare[nr_apariti[v[i]]-1]=v[i];
    --nr_apariti[v[i]];}*/
for (i = 0; v[i]; ++i) v[i] = afisare[i];}
```

3.5 Shell Sort

```
void ShellSort(int v[], int n) { //parcurg cu 2 for-uri fiecare interval
  for (int interval = n / 2; interval > 0; interval /= 2) {
    for (int i = interval; i < n; i++)
    { int aux = v[i]; int j;
      for (j = i; j >= interval && v[j- interval] > aux; j -= interval){
        v[j] = v[j - interval]; } v[j] = aux;} } }
```

3.6 Bubble Sort

```
void BubbleSort(int v[],int n){//parcurg cu for
pentru a accesa fiecare element al vectorul
for(int j = 0; j < (n-1); ++j){/*verific daca
interschimbarea se va face*/int ok = 0;
//parcurgem cu for pentru a compara doua elemente
    for (int i = 0; i < (n-j-1); ++i)
```

```
    {//compar doua elemente din vector
    daca schimbam  semnul > in < o sa sortam descrescator
if (v[i] > v[i + 1]) {//interschimbarea are loc daca
elementele nu sunt in ordinea dorita
int aux = v[i]; v[i] = v[i + 1]; v[i + 1] = aux;  ok = 1; } }
//daca nu interschimbam inseamna ca tabloul este deja sortat
    if (ok == 0) break; } }
Clasa de eficienta(complexitate):T(n)= Teta(n^2)~\cite{dobosiewicz1980efficient}
```

[4]

3.7 Merge Sort

```
// Imbinam cei doi subvectori a[] and b[] pentru a obitine v[]
void Imbinare(int v[], int c, int e, int f)
{ // a=P[c..e] si b=P[e+1..f]
  int l1 = e - c + 1; int l2 = f - e; int a[l1], b[l2];
  for (int i = 0; i < l1; i++) a[i] = v[c + i];
  for (int j = 0; j < l2; j++) b[j] = v[e + 1 + j];
  //retinem indicele curent al subvectorilor si al vectorului principal
  int i, j,  k; i = 0; j = 0; k = c;
//pana cand ajungem la sfarsitul fiecarui subvector A sau B
il vom alege pe cel mai mare
  //si ii vom pune la pozitia lor corecta in vectorul P[z...f]
  while (i < l1 && j < l2) { if (a[i] <= b[j])
  { v[k] = a[i]; i++; } else
  {v[k] = b[j]; j++; } k++; }
//cand ramanem fara elemente in subvectori
a[] sau b[] iau elementele ramase le pun P[c..f]
  while (i < l1) { v[k] = a[i]; i++; k++;}
  while (j < l2) { v[k] = b[j]; j++; k++; } }
void MergeSort(int v[], int l, int f) {if (l < f) { /*z este pozitia unde vectorul
este impartit in doi subvectori */  int z = l + (f - l) / 2;
    MergeSort(v, l, z);  MergeSort(v, z + 1, f); Imbinare(v, l, z, f); } }
```

[7]

4 Experiment/studiu de caz

Specicatiile calculatorului folosit sunt : Windows 10 64-bit operating system, cu un procesor Intel Core i7 si 8GB RAM.
In Tabel am comparat algoritmii pe baza cazului favorabil, cazului mediu si cazului defavorabil.
Experiment pentru 5000 de elemente:
Dupa cum putem observa in tabel, timpii de executie sunt relativ mici cand introducem 5000 de elemente. Algoritmii se comporta conform teoriei, singurele

exceptii sunt la InsertionSort care se descurca bine pe cazurile favorabile si QuickSort , care ofera rezultate mai slabe decat sortarea prin selectie.

5000 de elemente			
Numele algoritmului	Cazul favorabil	Cazul mediu	Cazul defavorabil
Bubble Sort	0.65sec	1.1 sec	1.5 sec
Sortarea rapida	1.1 sec	0.031 sec	0.073 sec
Sortarea prin selectie	0.47 sec	0.4 sec	0.5 sec
Sortarea prin Insertie	0.032 sec	0.60 sec	1.08 sec
Sortarea prin interschimbare	0.055 sec	0.055 sec	0.055 sec

Experiment pentru 10,000 de elemente:

Dupa cum putem vedea in tabel , rezultatele raman asemanatoare
Timpii de executie cresc exponential , apropiindu-se de valori
mult mai considerabile in comparatie cu cazul anterior unde toti algoritmii se executau in mai putin de 2 secunde. Singurul algoritm care inca ofera rezultate foarte bune este sortarea prin interclasare care, in toate cazurile, ofera timpi insesizabili.

10,000 de elemente			
Numele algoritmului	Cazul favorabil	Cazul mediu	Cazul defavorabil
Bubble Sort	2.7 sec	4.6 sec	5.59 sec
Sortarea rapida	4.41 sec	0.07 sec	0.18 sec
Sortarea prin selectie	1.91 sec	1.91 sec	1.98 sec
Sortarea prin Insertie	0.052 sec	2.29sec	4.44 sec
Sortarea prin interschimbare	0.07 sec	0.09 sec	0.069 sec

Experiment pentru 100,000 de elemente:

Dupa trecerea in ordinul sutelor de mii , algoritmii care se incadreaza in clasa de complexitate O (n^2)
ofera rezultate in ordinul minutelor. Dintre acestia algoritmul cu cele mai importante rezultate este SelectionSort,iar

cel care are cele mai slabe rezultate este BubbleSort
Pentru algoritmii care se incadreaza in clasa de complexitate O(nlog(n)),
MergeSort ramane destul de important cu rezultatele mai mici de o secunda,
iar QuickSort ruleaza greu cu datele de intrare favorabile si
nefavorabile, dar se descurca mult mai bine decat MergeSort
la sortarea elementelor aleatorii.

100,000 de elemente			
Numele algoritmului	Cazul favorabil	Cazul mediu	Cazul defavorabil
Bubble Sort	273 sec	491 sec	671 sec
Sortarea rapida	452 sec	0.54 sec	229 sec
Sortarea prin selectie	186 sec	198 sec8	214 sec
Sortarea prin Insertie	0.41 sec	241.2sec	481 sec
Sortarea prin interschimbare	4.6 sec	0.8 sec	0.68sec

Experiment pentru 500,000 de elemente:

MergeSort ramane cea mai consistenta metoda de sortare. In aceste teste
metodele de sortare care apartin clasei de complexitate $O(n^2)$
devin nefolosibile pentru toate cazurile,timpul de executare a acestora este de
ordinul orelor.

500,000 de elemente			
Numele algoritmului	Cazul favorabil	Cazul mediu	Cazul defavorabil
Bubble Sort	9843 sec	17346 sec	24900 sec
Sortarea rapida	19890 sec	3.19 sec	9530 sec
Sortarea prin selectie	7280 sec	8293 sec8	9440 sec
Sortarea prin Insertie	3.2 sec	9980 sec	20210 sec
Sortarea prin interschimbare	5.5 sec	3.95 sec	4.2 sec

Caz cuvinte si caractere				
Numarul de elemente	5000 de elemente	10000 de elemente	100000 de elemente	500000 de elemente
Bubble Sort	1.45 sec	5.55 sec	668sec	21455 sec
Sortarea rapida	0.06 sec	0.17 sec	10.47 sec	346 sec
Sortarea prin selectie	0.65 sec	2.33 sec	282 sec	7406 sec
Sortarea prin Insertie	0.71 sec	2.55 sec	340 sec	13541 sec
Sortarea prin interschimbare	0.03 sec	0.79 sec	0.68 sec	3.60 sec

5 Experiment

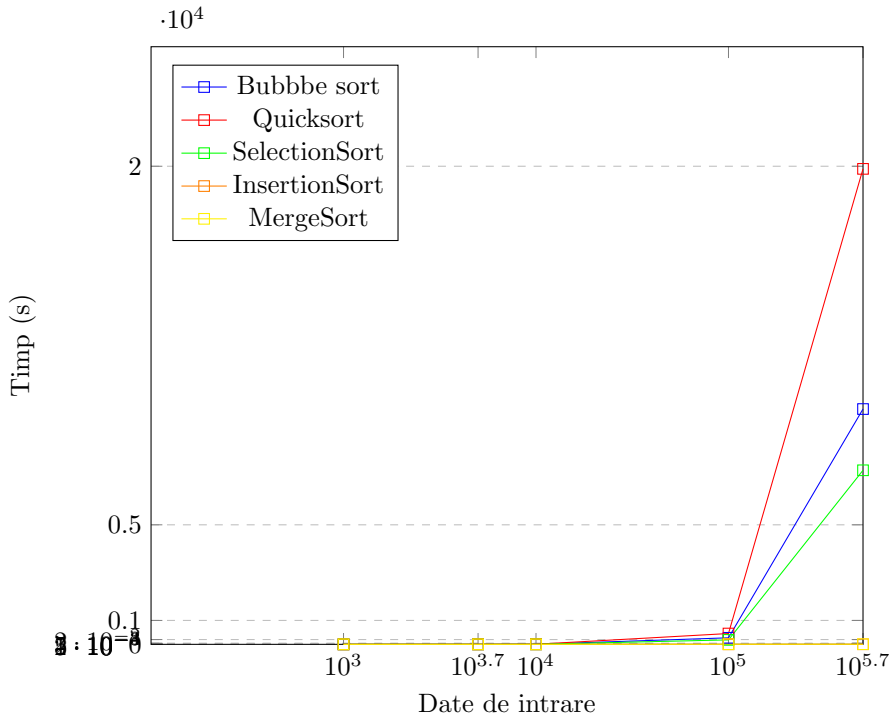

Figura 1: Comportamentul metodelor implementate pentru cazul favorabil(grafic semilogaritmic).

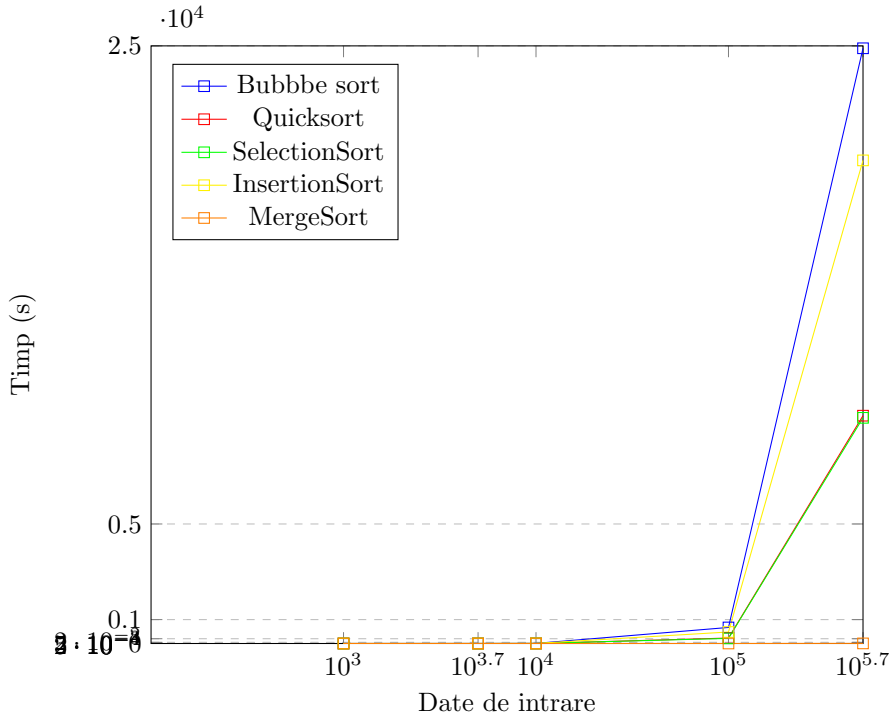

Figura 2: Comportamentul metodelor implementate pentru cazul defavorabil (grafic semilogaritmic).

6 Comparaţie cu literatura

Lucrarea lui Gabriel Balici începe prin a explica la ce sunt buni algoritmi de sortare și de ce este important sa utilizăm algoritmi eficienti. Ce este asemanator la lucrarea lui Gabriel Balici este ca el explica fiecare algoritm în parte așa cum am făcut și eu, de asemenea și el face teste pentru cazul favorabil ,mediu și defavorabil și notează rezultatele in tabel.
O diferenta intre lucrarea mea si a lui ar fi ca el nu a introdus în grafic toți algoritmi pe care ia studiat.O altă asemănare între lucrarea lui și a mea ar fi că și el a ajuns la aceeași concluzie ca Bubble Sort este cea mai ineficienta metoda de sortare.Cea mai rapidă metoda de sortare este QuickSort și ca QuickSort este mai rapid decat MergeSort.
Lucrarea lui Daniel-Bogdan Niculae începe prin introducerea conceptului de sortare .În finalul introducerii sunt explicate modalități posibile de sortare a unor seturi de obiecte, și proprietăți generale ale algoritmilor de sortare. În secțiunea 2 este luat în parte, și prezentat, fiecare algoritm de sortare folosit în partea practică a lucrării, specificand modul în care acesta sortează, cateva caracteristici teoretice precum complexitatea temporala, si un fragment de cod cu implementarea acestuia.O altă asemănare cu lucrarea mea ar fi partea experi-

mentală, unde fiecare algoritm este testat pe o multitudine de liste de elemente, acestea fiind alcătuite fie din numere, fie din caractere. Rezultatele acestui experiment sunt prezentate succint în concluzie, unde este specificat că Bubble sort are cea mai slabă performanță, urmat de Selection sort si Insertion sort, deși acestea din urma au anumite caracteristici favorabile în comparatie cu BubbleSort. Dintre algoritmii cu complexitatea temporala mai scăzută, autorul afirmă că Merge sort obține performanțe excelente indiferent de tipul datelor de intrare sau numărul lor, pe cand Quicksort are performante sub așteptări, exceptand situatia când lista de elemente este complet aleatorie. Drept urmare, prezentarea lucrării este suficient de clară.
Ce este asemanator la lucrarea lui Daniel-Bogdan Niculae fata de lucrarea mea este ca acesta are o concluzie detaliată.
Ce este diferit la lucrarea mea fata de lucrarea lui Daniel-Bogdan Niculae este ca acesta nu folosește grafice pentru a ușura citirea rezultatelor de către cititori.
Lucrarea lui Dan Salistean, autorul încearcă să motiveze importanța cunoașterii diferențelor dintre algoritmii de sortare, amintind de complexitatea spațială și de cea temporală. Sunt descrise cinci metode de sortare: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort și Quick Sort acestea fiind si metode studiate din lucrarea mea.
O comparatie intre lucrarea mea și lucrarea lui Dan Salistean ar fi ca acesta are rezultate posibil triviale în timp ce rezultatele mele sunt originale și de asemenea în descrierea algoritmilor apar greșeli.
De asemenea autorul a ajuns la o concluzie identică cu concluzia mea, QuickSort este cea mai eficientă metoda de sortare și BubbleSort este cea mai ineficienta metoda de sortare.

7 Concluzie

În această lucrare am prezentat o comparație între diverși algoritmi de sortare:Selection Sort,Insertion Sort,MergeSort,QuickSort și Bubble Sort. Am analizat performanța acestor algoritmi pentru același număr de date de intrare. Din testele făcute putem trage următoarele concluzii:
Când avem de sortat date de dimensiuni mari ar fi recomandat sa aplicăm QuickSort sau MergeSort.
BubbleSort: cea mai ineficienta metoda de sortare;
Pentru un număr mic de date de intrare, performanța algoritmilor prezentati este destul de apropiată, dar pentru un număr mare de date de intrare,Quick Sort este cel mai rapid algoritm,iar Selection Sort este cea mai lentă metoda de sortare. Sortarea prin Selectie: este utila cand e necesara sortare partiala (ex:se cauta primele k(k mai mic decat n) elemente in ordinecrescatoare).Sortarea prin Insertie: avantajoasă dacă tabloul este"aproape sortat".
Selection Sort este cel mai rapid dintre metodele de sortare de complexitate patratica în cazul în care avem un vector cu elemente sortate descrescator; (exceptand cazul în care avem un vector “aproape sortat”).
Pentru un vector “aproape sortat” sau sortat descescator Quick Sorteste de

doua ori (aproximativ) mai rapid decat Merge Sort.
Atat timp cat dimensiunea datelor de intrare este mica ,este irelevant ce metoda aplicăm. Sortare prin comparare în medie și în cel mai rău caz are aceeași complexitate de timp cu Selection Sort, Insertion Sort și Bubble Sort.Pentru a determina dacă un algoritm de sortare este bun,trebuie sa avem în vedere complexitatea în timp și alți factori de luat în considerare ar fi: gestionarea diferitelor tipuri de date,complexitatea codului și stabilitatea algoritmului. Din discuția de mai sus, putem concluziona că fiecare algoritm de sortare are anumite avantaje și dezavantaje în ceea ce privește utilizarea lui.Programatorii ar trebui sa aleaga algoritmul cel mai potrivit din punct de vedere al complexității în timp,gestiunea tipului de date utilizate și stabilitatea algoritmului ales. [3]

8 Bibliografie

Bibliografie

[1] Pooja Adhikari. Review on sorting algorithms a comparative study on two sorting algorithms. *Mississippi State, Mississippi*, 4, 2007.

[2] Alfred V Aho and John E Hopcroft. *The design and analysis of computer algorithms*. Pearson Education India, 1974.

[3] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[4] Wlodzimierz Dobosiewicz et al. An efficient variation of bubble sort. 1980.

[5] Andre Elisseeff, Theodoros Evgeniou, Massimiliano Pontil, and Leslie Pack Kaelbing. Stability of randomized learning algorithms. *Journal of Machine Learning Research*, 6(1), 2005.

[6] Iraj Hassanzadeh, MHS Afsari, and S Hassanzadeh. A new external sorting algorithm with selecting the record list location. *WSEAS Transactions on Communications*, 5(5):909, 2006.

[7] MA Kronrod. Optimal ordering algorithm without operational field. *DOKLADY AKADEMII NAUK SSSR*, 186(6):1256, 1969.

[8] Anchala Kumari and Soubhik Chakraborty. Software complexity: a statistical case study through insertion sort. *Applied mathematics and computation*, 190(1):40–50, 2007.

Daniela Zaharie,Introducere in proiectarea si analiza algoritmilor.
Algoritmi elementari de sortare. Timisoara, ISBN 9789736731211 9736731219, 2008.