- PROIECT -

Analiza Algoritmilor -AA-

Algoritmi de Sortare

Samoilescu Sebastian-Mihail

325CA

Facultatea de Automatica si Calculatoare

Universitatea Politehnica din Bucuresti

samoilescusebastian@gmail.com

9 Noiembrie 2019

Contents

1 Introducere

1.1 Descrierea problemei rezolvate

Ce reprezinta sortarea?

Ei bine, sortarea este reprezentata de aranjarea/rearanjarea unui set de date intr-o secventa ordonata pe baza unui criteriu bine definit.

De ce folosim sortarea?

In viata de zi cu zi nu exista garantia ca lucrurile se afla intr-o anumita ordine. Aproape orice informatie va aparea ca fiind aleatorie.
De asemenea, cu ajutorul sortarii putem rezolva sau optimiza alte probleme precum: cautarea de elemente, frecventa elementelor, gasirea celor mai apropiate elemente.

Cum realizam sortarea?

Aceasta este cea mai relevanta si importanta intrebare. Raspunsul, in mod evident, este depinde. Depinde de ce anume cunoastem despre datele ce urmeaza a fi sortate(daca sunt partial sortate, invers sortate, tipul datelor, etc).

1.2 Aplicatii Practice ale Algoritmilor de Sortare

Una dintre cele mai frecvente utilizari ale algoritmilor de sortare este in domeniul finantelor (bursei).
Peste 45 % din piata online de trading este realizata cu ajutorul acestor tip de algoritmi. Ei bine, acestia nu sunt utilizati in mod direct, ci vin in ajutorul altor algoritmi ce intra in alcatuirea procesului automatizat de comercializare(Algorithmic Trading)

1.3 Specificarea solutiilor alese

- ShellSort
- RadixSort
- TimSort

1.4 Criterii de evaluare pentru solutia propusa

Pentru a testa solutiile alese voi crea 3 grupe de teste:

- Date random
- Date aproape sortate
- Date invers sortate

Pentru fiecare grupa voi alege 5 subgrupe cu urmatorele dimensiuni:(100, 100.000, 500.000, 750.000, 1.000.000).
In cadrul testelor voi folosi date atat de tip Integer cat si String.

2 Prezentarea Solutiilor

2.1 Descrierea algoritmilor

2.1.1 Shell Sort

Shell Sort este un algoritm ce se incadreaza in categoria **in-place algorithm** (algoritmi ce modeleaza input - ul fara structuri de date auxiliare).
Shell Sort aranjeaza primele elemente ce se afla la o distanta h unele de altele, reducand succesiv intervalul h.
Putem observa ca **Shell Sort** nu este altceva decat o generalizare a lui **Insertion Sort**.
Performanta algoritmului depinde de secventa de intervale aleasa.

Cateva dintre cele mai utilizate intervale sunt:

- Shell's original sequence: N/2 , N/4 , . . . , 1
- Knuth's increments: 1, 4, 13, . . . , (3k – 1) / 2
- Papernov and Stasevich increment: 1, 3, 5, 9, 17, 33, 65,..

Implentari pentru creearea de gap-uri:

```cpp
void createNormalGaps(std::vector<int> &gaps, int dimension) {
    int k = dimension / 2;
    for (int i = k; i > 0; i/=2) {
        gaps.push_back(i);
    }
}
void createKnuthGaps(std::vector<int> &gaps, int dimension) {
    for (int i = 1; i <= dimension / 2; i=(i * 3) + 1) {
        gaps.push_back(i);
    }
    std::reverse(gaps.begin(), gaps.end());
}
 void createPrattGaps(std::vector<int> &gaps, int dimension) {
     int p = 0, q  = 0;
     int gap = 1;
     while (gap <= dimension) {
         gaps.push_back(gap);

     }
}
void createSedgewickGaps(std::vector<int> &gaps, int dimension) {
    int gap = 1;
    int i = 1;
    while(gap <= dimension / 2) {
        gaps.push_back(gap);
        gap = pow(4, i) + 3 * pow(2, i - 1) + 1;
        i++;
    }
}
```

Implementarea pentru ShellSort:

```
template<typename T>
void shellSort(std::vector<T> &array, std::vector<int> &gaps, int(*f)(T a, T b))
{   int dimension = array.size();
    for (int i = 0; i < gaps.size(); i++)
    {   int gap = gaps[i];
        for (int i = gap; i < dimension; i += 1)
        {
            T temp = array[i];
            int j;
            for (j = i; j >= gap && f(array[j - gap], temp) > 0; j -= gap)
                array[j] = array[j - gap];
            array[j] = temp;
        }
    }
}
```

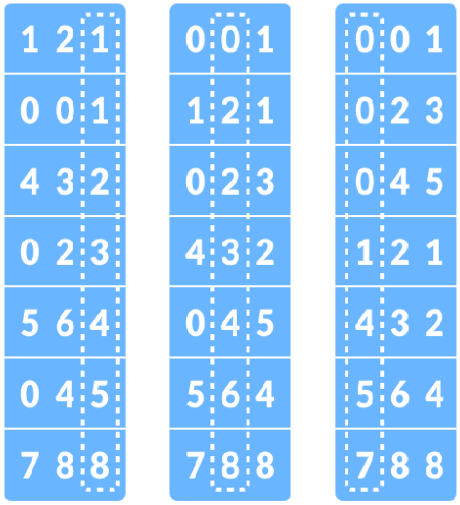
2.1.2 Radix Sort

Radix Sort este un algoritm ce se incadreaza in categoria **non-comparative sorting algorithms**. Acestea reuseste sa evita compararea elementelor grupand fiecare element intr-un container in functie de **radix - ul** propriu. Prin urmare, se poate realiza sortarea crescatoare/descrescatoarea in functie de bucket-uri.
Radix Sort poate fi aplicata doar pe date ce pot fi sortate **lexicografic** : numere, cuvinte.
Pentru a extinde functionalitatea algoritmului catre numere cu virgula ar trebui ca datele de intrarea se fie prelucrate in prealabil.
Trecerea numerelor reale spre numere intregi se va face prin inmultirea datelor cu 10^k unde k este numarul zecimalelor numerelor.
Cu toate acestea, exista limitari, intrucat numerele dupa prelucrare nu trebuie sa depaseasca lungimea unui long.

Mod de functionare pe tipul de date Integer

In primul pas, algoritmul sorteaza numerele dupa cifra unitatilor, apoi dupa cifra zecilor, pana la cifra cea mai semnificativa.

Exemplu:



Implementarea pentru tipul de date Integer:

```
void radixSort(std::vector<int> &array) {
    std::queue<int> digits[DIGITS];
    int length = array.size();
    int sizes[DIGITS] = {0};
    int negNumbers = 0;
    long digit_rank = 1, max_rank = 1;
    for(int i = 0; i < length; i++) {
        digits[0].push(array[i]);
    }
    sizes[0] = digits[0].size();
    while (sizes[0] != (length - negNumbers ) || digit_rank == 1) {
        for (int i = 0; i < DIGITS - 1; i++) {
            while(sizes[i] > 0) {
                int number = digits[i].front();
                digits[i].pop();
                if (abs(number) / digit_rank == 0 && number < 0) {
                    digits[10].push(number);
                    negNumbers++;
                } else {
                    int digit = (abs(number) / digit_rank) % MOD;
                    digits[digit].push(number);
                }
                sizes[i]--;
            }

        }
        digit_rank *= 10;
        for (int i = 0; i < DIGITS - 1; i++) {
            sizes[i] = digits[i].size();
        }
    }
    int negLength = digits[DIGITS - 1].size() - 1;
    int i = 0;
    while(negLength >= 0) {
        array[negLength--] = digits[DIGITS - 1].front();
        digits[DIGITS - 1].pop();
    }
    while(i < sizes[0]) {
        array[i + negNumbers] = digits[0].front();
        digits[0].pop();
        i++;
    }

}
```

Implementarea pentru tipul de date String:

```
int getMax(std::vector<std::string> strings){
    int max = strings[0].size();
    for (int i = 1; i < strings.size(); i++){
        if (strings[i].size() > max)
            max = strings[i].size();
    }
    return max;
}
```

```cpp
void countSort(std::vector<std::string> &strings, int position){
    int *frequency = NULL;
    int size = strings.size();
    std::string *new_order = new std::string[size];
    frequency = new int[CHAR_NUMBERS];


    for (int i = 0; i < CHAR_NUMBERS; i++){
        frequency[i] = 0;
    }
    for (int j = 0; j <size; j++){
        frequency[position < strings[j].size() ? strings[j][position] + 1 : 0]++;
    }

    for (int char_index = 1; char_index < CHAR_NUMBERS; char_index++){
        frequency[char_index] += frequency[char_index - 1];
    }

    for (int i = size - 1; i >= 0; i--){
        new_order[frequency[position < strings[i].size() ? strings[i][position] + 1 : 0] - 1]
                                                                    = strings[i];
        frequency[position < strings[i].size()   ? strings[i][position] + 1 : 0]--;
    }

    for (int i = 0; i < size; i++){
        strings[i] = new_order[i];
    }

    // eliberarea memoriei
    delete[] new_order;
    delete[] frequency;
}


void radixSort(std::vector<std::string> &b){
    int max = getMax(b);
    for (int digit = max; digit > 0; digit--){
        countSort(b, digit - 1);
    }
}
```

2.1.3 Timsort

Timsort este un algoritm de sortare **hibrid**, derivat din **Merge Sort** si **Insertion Sort**. Algoritmul a fost proiectat sa functioneze eficient pe diverse tipuri de date ce provin din lumea reala.
In cazul in care lista contine mai putin de 32 de elemente, sortarea va fi facute de un **Insertion Sort**. Altfel, algoritmul va cauta sa gaseasca secvente sortate (crescator sau descrescator), denumite **minrun**. Pentru ca merge - ul sa fie rapid, se va incerca sa se aleaga minrun - uri de o anumita lungime(lungimi egale) astfel incat numarul lor sa fie o putere a lui 2.
Merge ul sa realizeaza cu ajutorul unui **stack**. Dupa ce au fost indentificate minrun-uri acestea se pun in stack pentru a se face merge. Pentru a exploata cat mai mult minrun-urile ce vor veni in stack, algoritmul incearca sa le tina pe cele existente timp indelungat pana sa faca merge. Pe de alta parte, ele nu pot fi tinute prea in mult in stiva intrucat ar consuma prea multa memeorie.
Pentru a rezolva aceasta probleme, Timsort - ul introduce urmatoarea regula. Minrun - urile vor fi merge - uite cand(A,B,C - ultimele elemente din stiva):

- A + B > C
- B > C

In aceste conditii se va face merge intre B si C.

Galloping

Cat timp Timsort face merge pe A si B si observa ca un minrun are castig(un element din A/B este comparat de K ori, K o constanta aleasa). Daca se dovedeste ca minrun A are elemente mai mici decat B, atunci algoritmul nu va mai face merge si va cauta binar pe A[0] in B si B[j] in A pentru a cauta pozitia din B unde se va incepe copierea lui A si pozitia la care sa se opreasca copierea.

Implementare Tim Sort:

```cpp
template<typename type>
void insertionSort(std::vector<type> &array, int left, int right, int(*f)(type a, type b)) {
    for (int i = left + 1; i <= right; i++) {
        type temp = array[i];
        int j = i - 1;
        while (j >= left && f(array[j], temp) >= 0) {
            array[j + 1] = array[j];
            j--;
        }
        array[j + 1] = temp;
    }
}
template<typename type>
void merge(std::vector<type> &array, int left, int middle, int right, int(*f)(type a, type b)) {
    int len1 = middle - left + 1, len2 = right - middle;
    std::vector<type> left_array(len1), right_array(len2);
    for (int i = 0; i < len1; i++) {
        left_array[i] = array[left + i];
    }
    for (int i = 0; i < len2; i++) {
        right_array[i] = array[middle + 1 + i];
    }
    int i = 0;
    int j = 0;
    int k = left;
    while (i < len1 && j < len2)  {
        if (f(left_array[i],right_array[j]) <= 0) {
            array[k++] = left_array[i++];
        }
        else {
            array[k++] = right_array[j++];
        }
    }
    while (i < len1) {
        array[k++] = left_array[i++];
    }
    while (j < len2) {
        array[k++] = right_array[j++];
    }
}
```

```cpp
template <typename type>
void timSort(std::vector<type> &array, int(*f)(type a, type b)) {
    int dimension = array.size();
    // sortez secventele de lungime run
    for (int i = 0; i < dimension; i += RUN) {
        insertionSort(array, i, std::min((i + 31), (dimension - 1)), f);
    }

    // incep sa unesc secventele de lunigme 32
    // pentru a obtine lungime de 64, 128 ..
    for (int size = RUN; size < dimension; size = 2 * size) {
        // aleg inceputul secventei din stanga.
        // se vor uni secventele array[left..left+size-1] si array[left+size, left+2*size-1]
        // dupa fiecare unire vom creste marimea secventei ce va fi unita
        for (int left = 0; left < dimension; left += 2 * size) {
            // cautam finalul secventei din dreapta
            // mijlocul va reprezenta sfarsitul, respectiv inceputul celor doua secvente
            int middle = std::min((left + size - 1), (dimension - 1));
            int right = std::min((left + 2 * size - 1), (dimension - 1));
            merge(array, left, middle, right, f);
        }
    }
}
```

2.2 Analiza complextiatii solutiilor

2.2.1 ShellSort

Cel mai defavorabil caz

Cel mai defavorabil caz depinde de alegerea gap - ului si are drept **complexitate temporala O(n^2)**.
Numarul de comparatii realizate de algoritm pentru un anumit gap, g, este maxim $\frac{n^2}{g}$(costul temporal al celor doua for - uri din interior).
Pentru setul de gap - uri curent observam ca gap - ul la pasul k se obtine injumatatind gap - ul de la pasul k-1.
Complextiatea toala este data de suma:

$$S = n^2 + \frac{n^2}{2} + \frac{n^2}{4} + ...+ \frac{n^2}{2^h}$$
$$S = n^2(1 + \frac{1}{2} + \frac{1}{4} + ...+ \frac{1}{2^h})$$
$$S \leqslant 2n^2$$

Prin urmare S $\in$ O(n^2), deci cazul cel mai defavorabil este O(n^2).

Complexitati in functie de gap - ul ales

- Papernov and Stasevich, 1965 $\theta(n^{\frac{3}{2}})$
- Pratt, 1971 $\theta(Nlog^2N)$
- Sedgewick, 1982 $\theta(n^{\frac{4}{3}})$

Pentru gap-uri ale caror termeni sunt apropiati algoritmul va avea o performanta scazuta.
Exemplu:

- n-1, n-2, n-3...1
- n-3, n-6, n-9,...1

ShellSort vs BubbleSort

ShellSort functioneaza mai bine decat BubbleSort intrucat acesta sorteaza numere aflate la o distanta h. Daca vectorul este aproape sortat(2 elemente neordonate) un BubbleSort risca sa faca n interschimbari, in timp ce ShellSort - ul poate face o singura interschimbare in functie de gap-ul ales.

2.2.2 RadixSort

RadixSort este un algoritm de sortare liniara si functioneaza fara a compara elementele. In teorie, **complexitatea timpului mediu de rulare este O(d*N)**. In practica, eficienta lui depinde de valoarea lui d , unde d este numarul de cifre din numar sau de litere dintr - un cuvant. Daca numarul de cifre/litere este constant pentru toate valorile , atunci performanta RadixSort - ului este mai buna decat a altor algoritmi.

2.2.3 TimSort

- **Caz favorabil** - acesta este atins atunci cand setul de date este deja sortat. In acest caz, complexitatea algoritmului este de O(n).
- **Caz defavorabil** - in cel mai rau caz, timsort are o performanta **O(N*logN)** pentru a sorta setul de date. Acest lucru se datoreaza impartirii setului de date in mini-seturi si unirea acestora cu algoritmul de merge.

  Timsort - ul are performanta mai buna decat QuickSort deoarcere accesarea elementelor de catre TimSort este mai eficienta, intrucat spre deosebire de QuickSort, TimSort - ul sorteaza elementele din subsecventa ce sunt stocate liniar in memorie.

2.3 Avantaje si Dezavantaje pentru Solutiile alese

2.3.1 ShellSort

Avantaje	Dezavantaje
1. Algoritmul functioneaza in mod eficient pe seturi de date mici si medii.	1. Este un algoritm de sortare complex care nu este eficient precum heap, merge sau quick sort.
2. Are o performanta buna pe liste ce contin elemente duplicate.	2. Performanta acestuia depinde in mod direct de lista de gap - uri aleasa.

2.3.2 RadixSort

Avantaje	Dezavantaje
1. Este eficient atunci cand plaja de valori a setului de date este scurta.	1. Flexibilitate redusa comparativ cu alti algoritmi de sortare.
2. Performanta acestuia nu depinde de modul in care apar elementele in setul de date.	2. Costisitor din punctul de vedere al consumului de memorie.

2.3.3 TimSort

Avantaje	Dezavantaje
1. Capabil sa sorteze eficient date provenite din lumea reala.	1. Foloseste mai multa memorie aditionala decat principalii competitori.
2. Este un algoritm de sortare stabil.	

3 Evaluare

3.1 Construirea Testelor

- Pentru generarea testelor am folosit 6 surse scrise in limbajul de programare C++.
- Pentru a genera date random am folosit functia din C++ rand() pe o plaja diversa de valori pe care o corectam sau nu ulterior prin taiere de cifre.
- Datele aproape sortate au fost obtinute din date sortate pe baza carora s-au amestecat $< 10\%$.
- Pentru creeare testelor pe tipul de date string mi-am format un sir de caractere, dupa care generez indexi random pentru a crea alte siruri de caractere.
- Lungimea string - urilor este de maxim 30 de caractere.

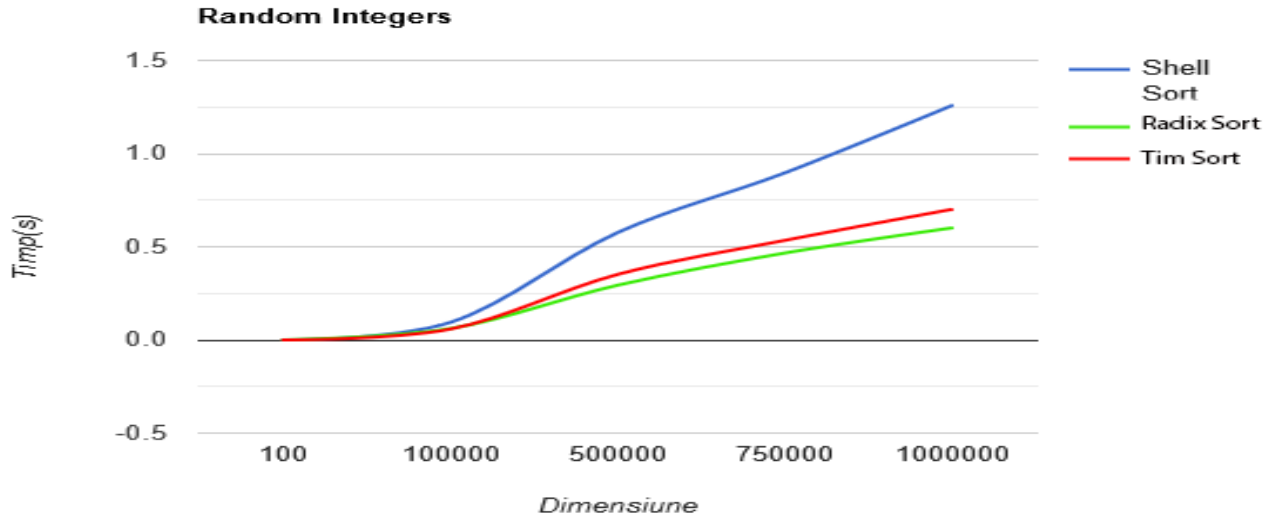
3.2 Specificatii sistem

- Sistem de operare: Ubuntu 18.04 64-bit
- Procesor: Intel® Core™ i5-8265U
- Memorie: 8 GB

3.3 Vizualizare performante

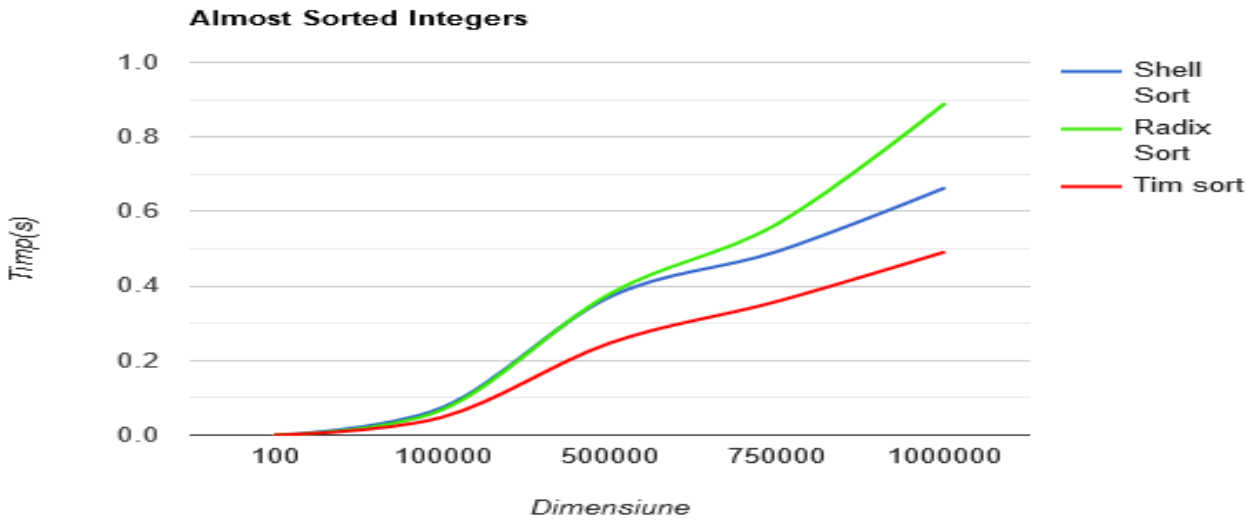
Numere generate random

	A	B	C	D
1	Dimensiune	Shell sort	Radix Sort	Tim Sort
2	100	3.1e-5	1.6e-4	3.3e-5
3	100.000	0.094	0.061	0.058
4	500.000	0.577	0.294	0.352
5	750.000	0.898	0.467	0.535
6	1.000.000	1.261	0.602	0.702



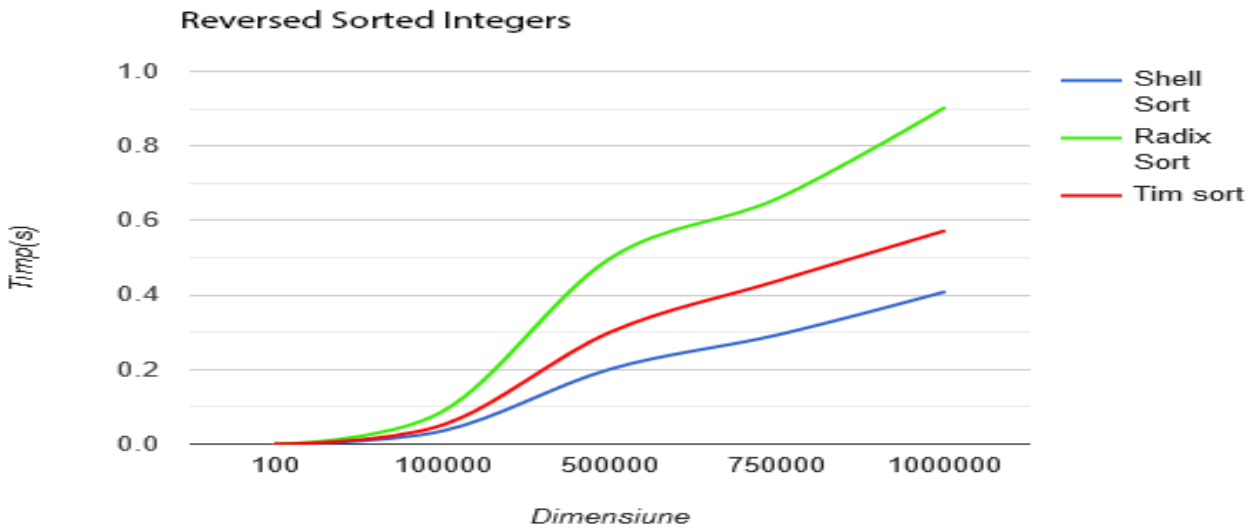
Numere aproape sortate

	A	B	C	D
1	Dimensiune	Shell sort	Radix Sort	Tim Sort
2	100	2.2e-5	1.0e-4	2.2e-5
3	100.000	0.070	0.068	0.046
4	500.000	0.371	0.379	0.247
5	750.000	0.493	1.046	0.359
6	1.000.000	0.664	1.091	0.492



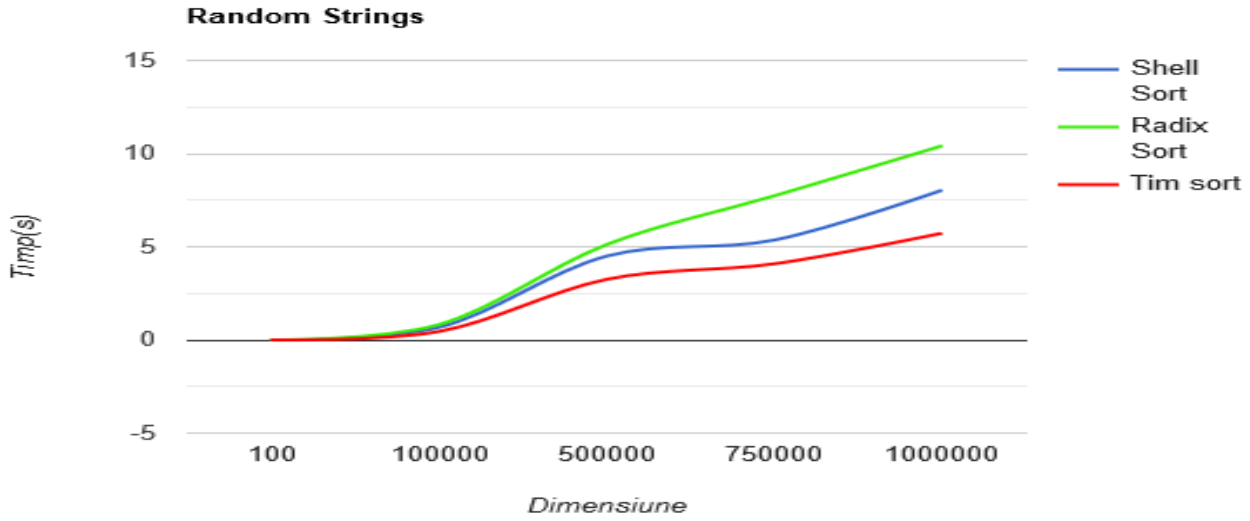
Numere sortate invers

	A	B	C	D
1	Dimensiune	Shell sort	Radix Sort	Tim Sort
2	100	1.4e-5	1.0e-4	4.7e-5
3	100.000	0.035	0.087	0.051
4	500.000	0.200	0.498	0.300
5	750.000	0.298	0.659	0.438
6	1.000.000	0.409	0.904	0.573



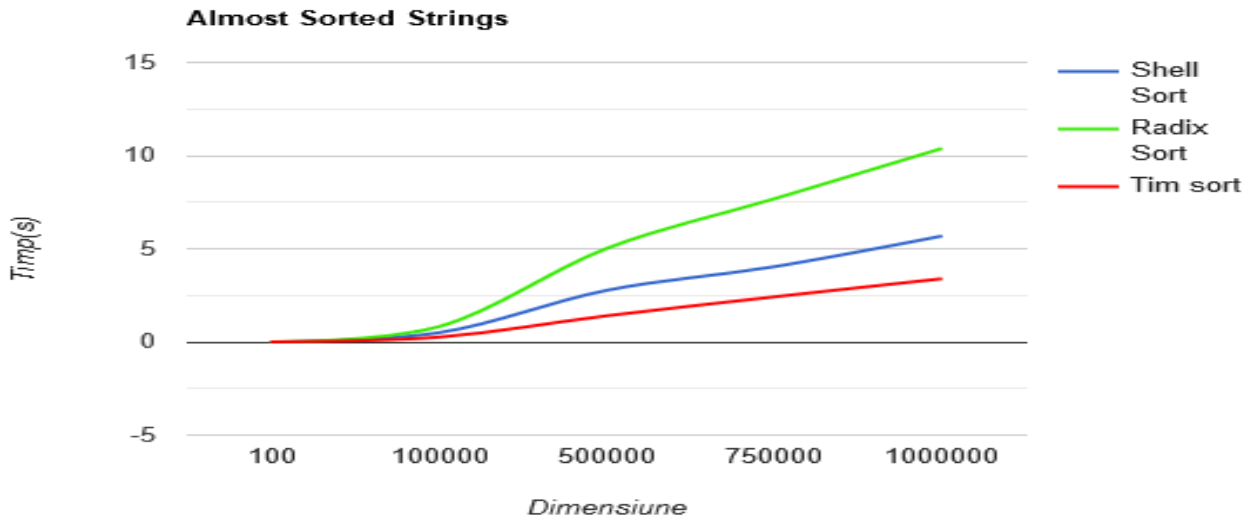
String - uri generate random

	A	B	C	D
1	Dimensiune	Shell sort	Radix Sort	Tim Sort
2	100	2.0e-4	5.6e-4	1.7e-4
3	100.000	0.682	0.826	0.453
4	500.000	4.512	5.123	3.254
5	750.000	5.362	7.737	4.086
6	1.000.000	8.042	10.42	5.729



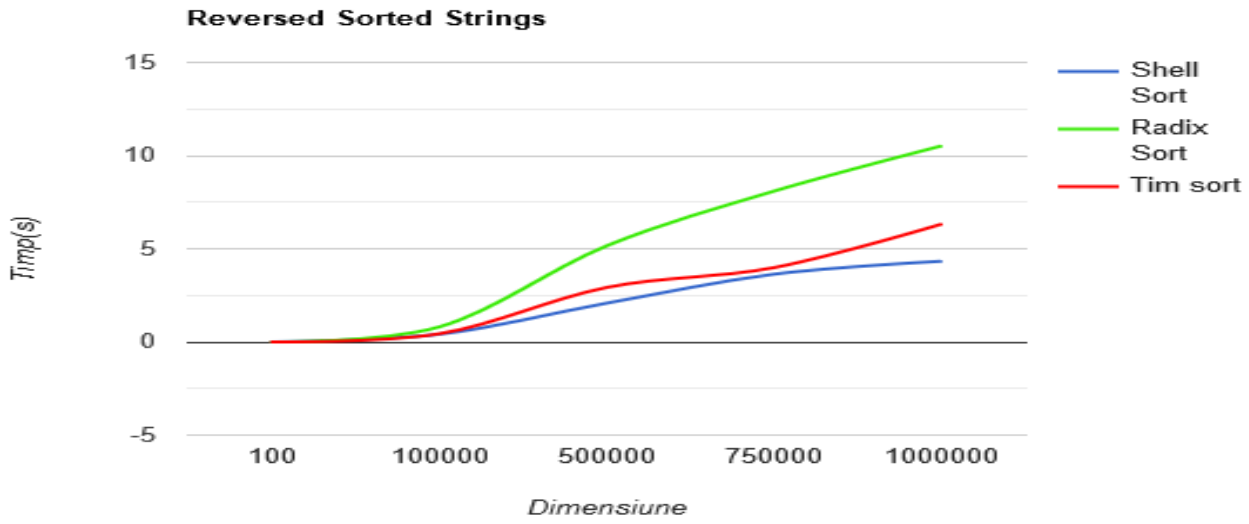
String - uri aproape sortate

	A	B	C	D
1	Dimensiune	Shell sort	Radix Sort	Tim Sort
2	100	1.2e-4	5.8e-4	1.7e-4
3	100.000	0.498	0.826	0.258
4	500.000	2.771	5.094	1.407
5	750.000	4.039	7.686	2.421
6	1.000.000	5.686	10.39	3.393



String - uri invers sortate

	A	B	C	D
1	Dimensiune	Shell sort	Radix Sort	Tim Sort
2	100	1.1e-4	6.8e-4	2.8e-4
3	100.000	0.412	0.806	0.437
4	500.000	2.093	5.166	2.926
5	750.000	3.647	8.101	3.993
6	1.000.000	4.333	10.53	6.634



3.4 Interpretare rezultate

Cu ajutorul reprezentarilor grafice putem observa:

- Radix Sort reuseste sa aiba o performanta mai buna doar pe testele generate random, intrucat solutia nu necesita compararea elementelor.
- Shell Sort are o performanta buna pentru seturile de teste invers sortate.
- Tim Sort reuseste sa aiba o buna performanta pe toate tipurile de teste.

4 Concluzii

Asadar, nu exista o solutie magica cand vine vorba de problema sortarii.
Raspunsul problemei este unul des intalnit in lumea tehnologiei,si anume....depine. Depinde de cum arata setul de teste, memoria sistemului sau puterea acestuia de procesare.
Pentru un set de date de dimensiuni mici poate ar fi de ajuns un algoritm de tip insertionSort, pentru un set de date ce sunt invers sortate un algoritm de tip shellSort, etc.
Cu toate acestea, se pare ca timSort reuseste sa acopere o arie mai mare de cazuri practice, o consecinta a faptului ca a fost realizat in urma experientelor practice si nu teoretice.

5 Bibliografie

- https://en.wikipedia.org/wiki/Shellsort
- https://en.wikipedia.org/wiki/Radix_sort
- https://en.wikipedia.org/wiki/Timsort
- https://medium.com/@rylanbauermeister/understanding-timsort-191c758a42f3
- https://www.infopulse.com/blog/timsort-sorting-algorithm/
- https://hackernoon.com/timsort-the-fastest-sorting-algorithm-youve-never-heard-of-36b28417f399