Analiza Algoritmilor - TEMA 1
Etapa a III-a

Cernea David Andreas

December 2018

1 1.Introducere

1.1 Descrierea problemei rezolvate

Pentru aceasta tema am ales sa abordez tema sortarii. Ne propunem sa avem un vector de n elemente random , pe care , la sfarsit, folosind un algoritm de sortare, sa-l afisam cu elementele in ordine crescatoare.

1.2 Exemple de aplicatii prcatice pentru problema aleasa

Aplicatii practice ale acestei probleme o reprezinta sortarile in functie de un anumit criteriu de pe site-urile specializate pe vanzarea de bunuri ("ordoneaza dupa pret/popularitate/etc."), care vin in folosul clientilor, si, totodata, sortarile din bazele de date ale furnizorilor, care fac posibila mentinerea unei ordini in randul produselor de pe stoc.

1.3 Specificarea solutiilor alese

Solutiile alese sunt:

- Heap Sort
- Radix Sort
- Shell Sort

Heap Sort este un algoritm de sortare ce se foloseste de un arbore binar de tip max-heap.Formam max-heapul pe baza vectorului input, facem swap intre radacina si ultimul nod, stergem nodul si reluam rationamentul pana epuizam toate elementele.
Radix Sort este un algoritm de sortare ce se bazeaza pe un alt algoritm numit Counting Sort. Acesta se foloseste de un interval si implicit de un vector de frecventa pentru a sorta elementele. Radix Sort, pe de alta parte, completeaza Counting Sortul prin eficientizarea procesului aducand complexitatea de la n^2 la una liniara, facand sortarea tinand cont de cifrele elementelor.

Shell Sort este un algoritm de sortare cu salt (la inceput mare, dar cu fiecare iteratie devine din ce in ce mai mic). Shell Sort este o variatie a Insertion Sort, algoritm care pune elementele pe rand cat mai in stanga posibil, in ordine.

1.4 Specificarea criteriilor de evaluare alese pentru validarea solutiilor

Criteriul principal de evaluare ales pentru validarea solutiilor este corectitudinea sortarii. S-a folosit un program de sortare online pentru a compara solutia noastra cu cea reala cu scopul testarii corectitudinii programului nostru.
`https://www.online-utility.org/text/sort.jsp`

Alte criterii de avaluare:

- Numar elemente redus
- Elementele sunt aproape sortate
- Cazul cel mai probabil
- Cazul cel mai nefavorabil
- Cel mai putin cod scris
- s.a

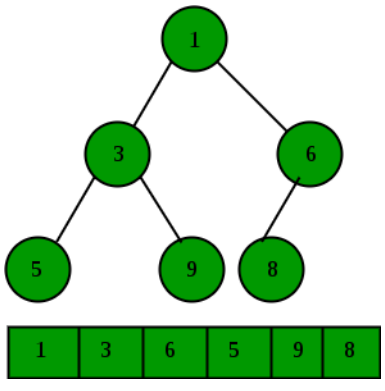
2 Prezentarea solutiilor

2.1 Descrierea modului in care functioneaza algoritmii alesi

2.1.1 Heap Sort

Heap Sort este un algoritm de sortare ce se foloseste de un arbore binar de tip max-heap.

Am ales sa reprezint arborele binar sub forma de Array intru-cat este eficient din punct de vedere al spatiului. Daca nodul parinte este retinut de indexul i, copilul stang va fi retinut in $i * 2 + 1$, iar copilul drept in $i * 2 + 2$ (luam in considerare faptul ca indexarea incepe de la 0);



Binary Heap Representation

Pasii algoritmului:

- Contruieste un MaxHeap pe baza fisierului de input
- Acum cel mai mare element este nodul radacina
- Interschimba radacina cu ultimul nod din MaxHeap si decrementeaza lungimea heap-ului
- Repeta pasii de mai sus cat timp lungimea MaxHeap-ului este mai mare decat 1

FORMAREA HEAP

```
// formeaza heap
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);
```

```
void heapify(std::vector<int>& arr, int n, int i){

    int largest = i; // cel mai mare e initial radacina
    int left = 2*i + 1; // frunza din stanga
    int right = 2*i + 2; // frunza din dreapta

    // daca frunza din stanga este mai mare ca radacina
    if (left < n && arr[left] > arr[largest])
        largest = left;

    // daca frunza din dreapta e mai mare ca radacina
    if (right < n && arr[right] > arr[largest])
        largest = right;

    // daca cel mai mare nu este radacina
    if (largest != i)
    {
        std::swap(arr[i], arr[largest]);

        // formeaza heap din subtree
        heapify(arr, n, largest);
    }
}
```

SORTAREA PROPRIU-ZISA

```
/ extrage cate un element din heap
    for (int i=n-1; i>=0; i--)
    {
        // muta radacina la capatul heap-ului
        std::swap(arr[0], arr[i]);

        // formeaza heap din heap-ul redus
        heapify(arr, i, 0);
    }
```

2.1.2 Radix Sort

Radix Sort este un algoritm bazat pe Counting Sort. Counting Sort are o complexitate de $O(n + k)$ pentru elemente din range-ul $0, k$. Ce se intampla cand elementeles sunt in range-ul $0, n^2$. Algoritmul devine extrem de ineficient, cu un timp de executie mai prost decat algoritmii bazati pe comparare(Merge Sort, Quick Sort, Heap Sort etc.). Aici intervine Radix Sort-ul. Acesta isi propune folosirea Counting Sortului insa pe fiecare cifra a elementelor incepand de la cea mai nesemnificativa pana la cea mai semnificativa. Astfel, timpul de executie devine unul liniar. Pasii algoritmului:

- Sorteaza vectorul de input folosind Counting Sort-ul pentru cifra de pe pozitia i
- Fa pasul de mai sus pentru fiecare cifra i, unde i incepe de la cea mai nesemnificativa cifra pana la cea mai semnificativa

Counting Sort:

- Se construieste un count array (de la 0 la 9) care numara aparitiile cifrelor
- Se modifica count array-ul astfel incat $array[i] += array[i-1]$
- Acum valorile din count array sunt indexii la care trebuie puse elementele din vectorul input
- Se pun intr-un vector de output valorile din vectorul input pe pozitiile aratate de count array. Dupa fiecare pozitionare se decrementeaza valoarea din count array
- Se copiaza vectorul output in vectorul input

Counting Sort

```
void countSort(std::vector<int>& arr, int n, int exp)
{
    int output[n];
    int i;
```

```
    int count[10] = {0}; // init cu 0

    // completeaza vectorul de pozitii
    for (i = 0; i < n; i++)
    {
        count[ (arr[i]/exp)%10 ]++;
    }
    // modifica count array-ul
    // la fiecare index va fi pozitia lui
    // din output array
    for (i = 1; i < 10; i++)
    {
        count[i] += count[i - 1];
    }
    // creeaza output arrayul
    // pune arr[i] la indexul = val din
    // count array - 1
    for (i = n - 1; i >= 0; i--)
    {
        output[count[ (arr[i]/exp)%10 ] - 1] = arr[i];
        count[ (arr[i]/exp)%10 ]--;
    }

    // copiaza output array-ul in array-ul initial
    for (i = 0; i < n; i++)
        arr[i] = output[i];
}
```

SORTAREA PROPRIU-ZISA

```
void sort(std::vector<int>& arr, int n, const bool reverse)
{
    // Cauta elementul cel mai mare ca sa stim maximul de cifre
    auto m = std::max_element(std::begin(arr), std::end(arr));
    // fa countingSort pentru fiecare cifra a numarului m
    for (int exp = 1; *m/exp > 0; exp *= 10)
    {
        countSort(arr, n, exp);
    }
}
```

2.1.3 Shell Sort

Algoritmul Shell Sort este o aditie la Insertion Sort, unde elementele se mutau cu o pozitie in fata pentru a face loc elementului curent pe pozitia sa naturala

pentru un vector sortat. Shell Sort aduce ideea de gap(salt). Astfel, ca nu mai comparam elemente situate pe pozitii apropiate, ci comparam elemente situate pe pozitii la gap distanta una fata de cealalta. dupa fiecare sortare partiala, se micsoreaza gap-ul. Cand se ajunge la gap = 0, vectorul este sortat.

Implementare

```
void sort(std::vector<int>& arr, int n, const bool reverse)
{
    for (int gap = n/2; gap > 0; gap /= 2)
    {
        // luam gapurile ca jumatati din numarul de elemente
        // se stie ca elem de la [0 ... gap - 1] sunt sortate
        // adauga succesiv cate un element pana vectorul este
        // sortat
        for (int i = gap; i < n; i += 1)
        {
            // adauga a[i] alaturi de elementele sortate
            // salveaza a[i] cu temp si fa loc pe pozitia i
            int temp = arr[i];

            // fa shift la dreapta elementelor sortate pana la locatia
            // lui a[i]
            int j;
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
                arr[j] = arr[j - gap];

            //  pune temp-ul pe pozitia care trebuie
            arr[j] = temp;
        }
    }

    if (reverse) {
        std::reverse(arr.begin(), arr.end());
    }
}
```

2.2 Analiza complexitatii algoritmilor

2.2.1 Heap Sort

Complexitatea algoritmului Heap Sort este constanta pentru toate cele trei cazuri (Best Case, Average Case, Worst Case), si anume $nlog(n)$. De ce?

Inaltimea unui Binary Tree Complet este $log(n)$ pentru n elemente. Pentru a face Heapify unui element al caror subtree-uri sunt deja MaxHeap-uri se fac

in **cel mai rau caz** log(n) pasi (ducem elementul de la radacina pana la ultimul nivel). Aceasta operatie se face de $n/2$ ori, deci complexitatea finala este $(n/2) * log(n) -> nlog(n)$
Complexitate lui Heap Sort poate fi $O(n)$ cand toate elementele sunt egale.

2.2.2 Radix Sort

Presupunem ca cel mai mare numar din vector are d cifre. Radix Sort se rezolva in $O(d * (n + b))$, unde b este baza de reprezentare a numerelor, de exemplu decimale, b este 10. Vrem sa aflam cat este d. Daca k ar fi cel mai mare numar din vector, $d = O(log_b(n))$. Complexitatea este mai mare decat cea a algoritmilor bazati pe comparatie. Pentru a trece din aceasta complexitate in una liniara b ar trebui sa fie egal cu n, complexitatea devenind $O(n)$. Cu alte cuvinte putem sa sortam un sir de la 1 la n^c daca numere sunt reprezentate in baza n (adica fiecare cifra ocupa $log_2(n)$ biti).

2.2.3 Shell Sort

Complexitatea acestui algoritm nu tine doar de cate numere vrem sa sortam, ci si de felul in care alegem gap-ul. Astfel, pentru un gap $\frac{n}{2k}$ complexitatea este $O(n^2)$. In acest exemplu, saltul se injumatateste cu fiecare pas.

OEIS	General term ($k \geq 1$)	Concrete gaps	Worst-case time complexity	Author and year of publication
	$\left\lfloor \frac{N}{2^k} \right\rfloor$	$\left\lfloor \frac{N}{2} \right\rfloor, \left\lfloor \frac{N}{4} \right\rfloor, \ldots, 1$	$\Theta\left(N^2\right)$ [e.g. when $N = 2^p$]	Shell, 1959[4]
	$2\left\lfloor \frac{N}{2^{k+1}} \right\rfloor + 1$	$2\left\lfloor \frac{N}{4} \right\rfloor + 1, \ldots, 3, 1$	$\Theta\left(N^{\frac{3}{2}}\right)$	Frank & Lazarus, 1960[8]
A168604	$2^k - 1$	$1, 3, 7, 15, 31, 63, \ldots$	$\Theta\left(N^{\frac{3}{2}}\right)$	Hibbard, 1963[9]
A083318	$2^k + 1$, prefixed with 1	$1, 3, 5, 9, 17, 33, 65, \ldots$	$\Theta\left(N^{\frac{3}{2}}\right)$	Papernov & Stasevich, 1965[10]
A003586	Successive numbers of the form $2^p 3^q$ (3-smooth numbers)	$1, 2, 3, 4, 6, 8, 9, 12, \ldots$	$\Theta\left(N \log^2 N\right)$	Pratt, 1971[1]
A003462	$\frac{3^k - 1}{2}$, not greater than $\left\lceil \frac{N}{3} \right\rceil$	$1, 4, 13, 40, 121, \ldots$	$\Theta\left(N^{\frac{3}{2}}\right)$	Pratt, 1971[1]
A036569	$\prod_I a_q$, where $a_q = \min\left\{n \in \mathbb{N} : n \geq \left(\frac{5}{2}\right)^{q+1}, \forall p : 0 \leq p < q \Rightarrow \gcd(a_p, n) = 1\right\}$ $I = \left\{0 \leq q < r \mid q \neq \frac{1}{2}\left(r^2 + r\right) - k\right\}$ $r = \left\lfloor \sqrt{2k + \sqrt{2k}} \right\rfloor$	$1, 3, 7, 21, 48, 112, \ldots$	$O\left(N^{1+\sqrt{\frac{8\ln(5/2)}{\ln(N)}}}\right)$	Incerpi & Sedgewick, 1985,[11] Knuth[3]
A036562	$4^k + 3 \cdot 2^{k-1} + 1$, prefixed with 1	$1, 8, 23, 77, 281, \ldots$	$O\left(N^{\frac{4}{3}}\right)$	Sedgewick, 1986[6]
A033622	$\begin{cases} 9\left(2^k - 2^{\frac{k}{2}}\right) + 1 & k \text{ even}, \\ 8 \cdot 2^k - 6 \cdot 2^{(k+1)/2} + 1 & k \text{ odd} \end{cases}$	$1, 5, 19, 41, 109, \ldots$	$O\left(N^{\frac{4}{3}}\right)$	Sedgewick, 1986[12]
	$h_k = \max\left\{\left\lfloor \frac{5h_{k-1}}{11} \right\rfloor, 1\right\}, h_0 = N$	$\left\lfloor \frac{5N}{11} \right\rfloor, \left\lfloor \frac{5}{11}\left\lfloor \frac{5N}{11} \right\rfloor \right\rfloor, \ldots, 1$	Unknown	Gonnet & Baeza-Yates, 1991[13]
A108870	$\left\lceil \frac{1}{5}\left(9 \cdot \left(\frac{9}{4}\right)^{k-1} - 4\right) \right\rceil$	$1, 4, 9, 20, 46, 103, \ldots$	Unknown	Tokuda, 1992[14]
A102549	Unknown (experimentally derived)	$1, 4, 10, 23, 57, 132, 301, 701$	Unknown	Ciura, 2001[15]

2.3 Prezentarea principalelor avantaje si dezavantaje pentru solutiile luate in considerare

Heap Sort

Avantaje:

- Eficienta in timp (daca anumiti algoritmi cresc exponential, Heap Sort creste logaritmic)
- Eficienta in spatiu (spatiu pe stiva alocat minimal)
- Consistenta (aceeasi complexitate - Best, Average, Worst Case)

Dezavantaje:

- Instabila (Poate schimba ordinea relativa e elementelor din vectorul de input)
- Slab pentru seturi de date extrem de mari

Radix Sort

Avantaje:

- Rapid pentru tipuri de date mici (unsigned int)
- Rapid pentru sortari de string-uri
- Rapid daca elementele sunt puse intr-o ordine descrescatoare

Dezavantaje:

- Nu atat de eficient ca alti algoritmi din punct de vedere al spatiului folosit
- Trebuie rescris pentru tipuri de date diferite

Shell Sort

Avantaje:

- Algoritm mai rapid decat Insertion Sort
- Interschimbare eficienta atunci cand elementele se afla la distanta una de cealalta

Dezavantaje:

- Nu la fel de eficient ca Merge, Heap sau Quick Sort

3 Evaluare

3.1 Descrierea modalitatii de construire a setului de teste folosite pentru validare.

- Testele 0, 1, 2, 3 cuprind numere random cuprinse intre 0 si 10 in numar de 100.000, 10.000, 1000, 10;
- Testul 4 cuprinde numere ordonate descrescator de la 0 la 9999 -¿ 10.000 de numere;
- Testul 5 cuprinde numere ordonate descrescator de la 0 la 99 -¿ 100 de numere;
- Testul 6 cuprinde numere de la 0 la 99 pe jumatate sortat crescator -¿ 100 de numere;
- Testul 7 cuprinde numere de la 0 la 9999 pe jumatate sortat crescator -¿ 10.000 de numere;
- Testul 8 cuprinde numere de la 0 la 9999 sortat crescator -¿ 10.000 de numere;
- Testul 9 cuprinde numere de la 0 la 9 sortat crescator -¿ 10 de numere;

3.2 Mentionati specificatiile sistemului de calcul pe care ati rulat testele (procesor, memorie disponibila

Procesor:

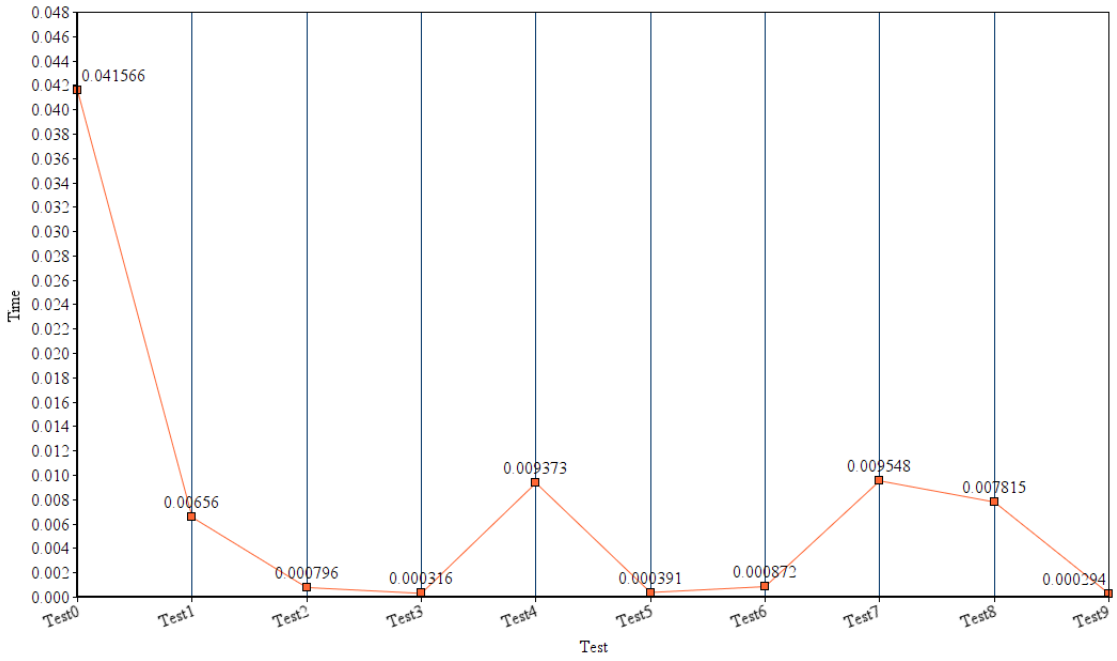
```
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              8
On-line CPU(s) list: 0-7
Thread(s) per core:  2
Core(s) per socket:  4
Socket(s):           1
NUMA node(s):        1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               142
Model name:          Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
Stepping:            10
CPU MHz:             800.033
CPU max MHz:         3400,0000
CPU min MHz:         400,0000
BogoMIPS:            3600.00
Virtualization:      VT-x
L1d cache:           32K
L1i cache:           32K
L2 cache:            256K
L3 cache:            6144K
NUMA node0 CPU(s):   0-7
```

Memorie libera:

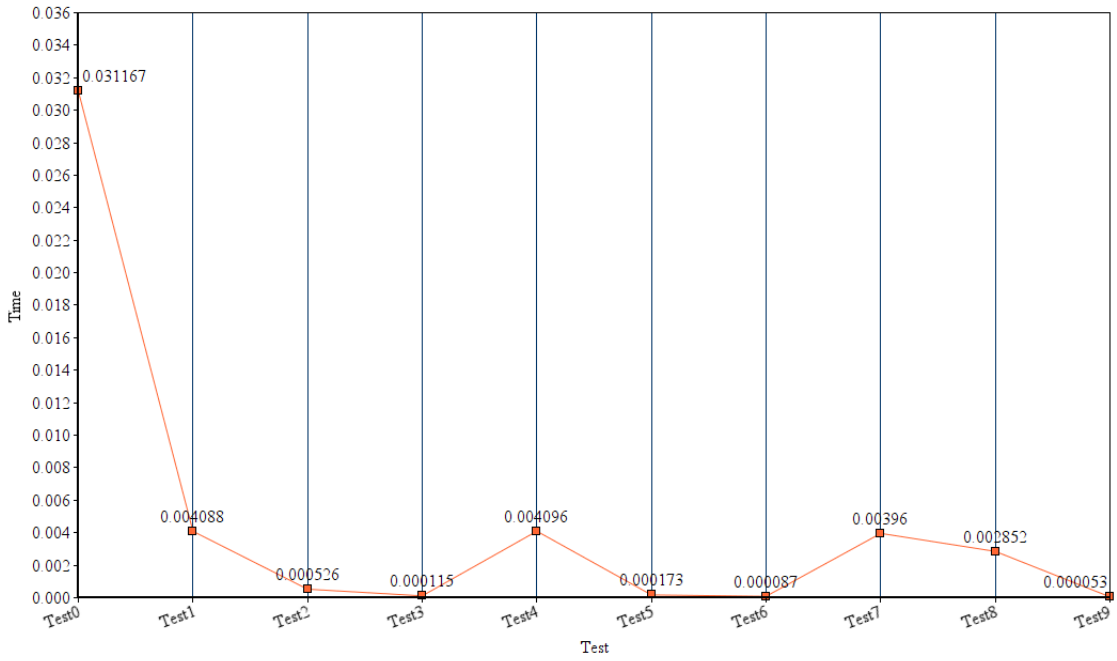
```
               total        used        free      shared  buff/cache   available
Mem:            3819        2797         182         361         838         431
Swap:           4470        1902        2568
```

3.3 Ilustrarea, folosind grafice/tabele, a rezultatelor evaluarii solutiilor pe setul de teste

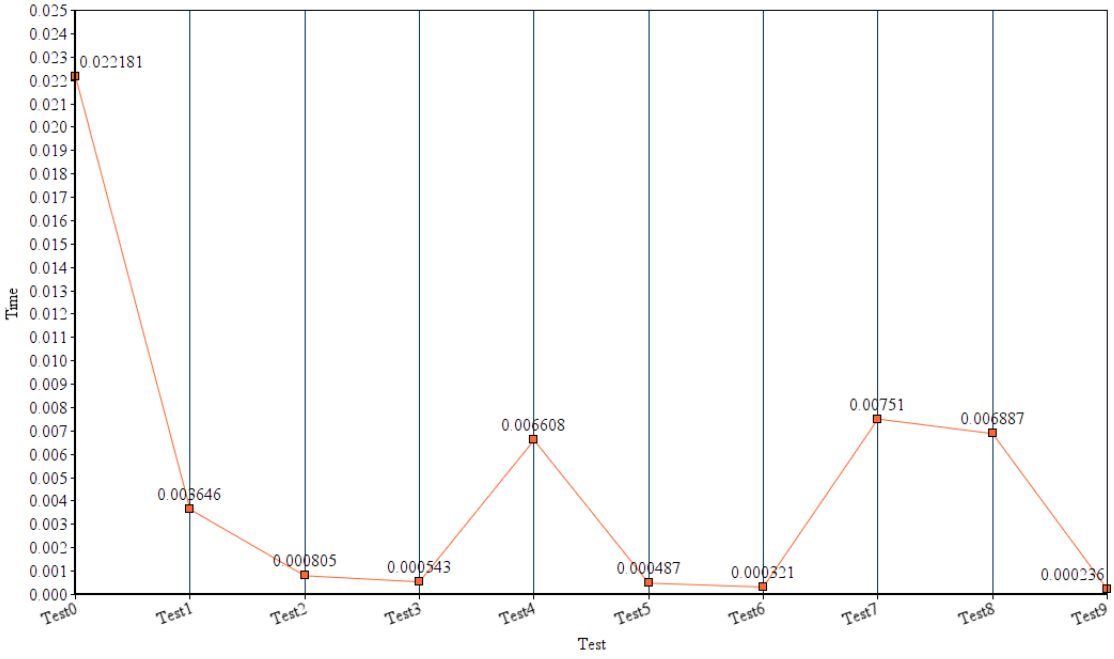
Heap Sort



Shell Sort


Time
Test
0.036
0.034
0.032
0.030
0.028
0.026
0.024
0.022
0.020
0.018
0.016
0.014
0.012
0.010
0.008
0.006
0.004
0.002
0.000
0.031167
0.004088
0.000526
0.000115
0.004096
0.000173
0.000087
0.00396
0.002852
0.000053
Test0
Test1
Test2
Test3
Test4
Test5
Test6
Test7
Test8
Test9

Radix Sort



3.4 Prezentarea succinta a valorilor obtinute pe teste

In cazul testelor cu un numar mare de date (Test0, Test1) generate random (100000, 10000), cel mai eficient algoritm este Radix Sort.

In cazut testelor cu numar mic de date (Test2, Test3) generate random (1000, 10), cel mai eficient algoritm este Shell Sort.

In cazut testelor cu numar mare si mic de date (Test4, Test5) ordonate descrescator (10000, 10), cel mai eficient algoritm este Shell Sort.

In cazut testelor cu numar mic si mare de date (Test6, Test7) pe jumatate ordonate crescator (10000, 100), cel mai eficient algoritm este Shell Sort.

In cazul testelor cu numar mare si mic de date (Test8, Test9) ordonate crescator (10000, 10), cel mai eficient algoritm este Shell Sort.

Per total, cel mai eficient algoritm de sortare este Shell Sort.

3.5 Bibliografie

References

[1] Gary Pollice, Stanley Selkow, George T. Heineman *Algorithms in a Nutshell.*

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein *Introduction to Algorithms.* (The MIT Press) Vol.3

[3] A Computer Science portal for geeks.
`http://www.geeksforgeeks.org/heap-sort/`
`https://www.geeksforgeeks.org/radix-sort/`
`https://www.geeksforgeeks.org/shellsort/`
`https://www.geeksforgeeks.org/stability-in-sorting-algorithms/`

[4] Graph Maker
`https://www.onlinecharttool.com/graph`

[5] Wikipedia
`https://en.wikipedia.org/wiki/Heapsort`
`https://en.wikipedia.org/wiki/Best, worst and average case`