Analiza Algoritmilor

-Sortari-

Aurtor : RATA Andrei 323CD

1 Introducere

1.1 Descrierea problemei rezolvate

Sortarile ,in general, au acelasi scop: punerea intr-o anumita ordine a unor elemente de un anumit tip specific.(Ex : ordonarea notelor unor elevi in ordine crescatoare). Insa, intervine problema urmatoare: exista posibilitatea ca datele noastre de intrare sa fie foarte mari sau foarte mici..De asemenea, pot exista cazuri cand datele de intrare pot fi ,,aproape sortate"(Sorting "almost sorted" lists.) s.a.m.d . Avand in vedere aceste situatii, algoritmi existenti de sortare obtin o complexitate adecvata(mai mica de O(n2)) in anumite situatii. Problema se pune astfel : ,, Cum as putea sorta, cat mai eficient, un sir de date aleatoriu ? **Sample Heading (Third Level).** Only two levels of headings should be numbered. Lower level headings remain unnumbered; they are formatted as run-in headings.

1.2 Exemple de aplicatii practice pentru problema aleasa

De multe ori, avem nevoie de o ierharizare a unor date, insa, daca datele noastre sunt de dimensiuni foarte mari (ex: date mai mari de 10.000), un algoritm de sortare ne poate rezolva problema intr-un timp rezonabil.Cateva exemple de astfel de situatii sunt ordinea elevilor dupa medie,ordinea angajatilor dupa salariu etc.

1.3 Specificarea solutiilor alese.

In aceasta lucrarea vor fi tratati 3 algoritmi de sortare de complexitati distincte ce lucreaza cu tipuri de date de diferite dimensiuni. Prin aceasta comparatie se va stabili aplicabilitatea acestora.

Se da un sir de N numere si se pune problema sortarii acestora in ordine crescatoare. Inputurile au diferite proprietati care ne ajuta in alegerea algoritmului cu performantele cele mai bune.

Daca datele noastre de intrare sunt repetitive (ex: 1 4 13 1 4 14), se foloseste CountingSort. Acest algoritm creeaza un vector de frecventa arrayFreq al elementelor date ca input. Odata creat, se parcurge vectorul si se insumeaza fiecare element cu frecventa precedenta,. Se creeaza apoi un vector newArray cu dimensiunea egala cu frecventa maxima.In final,se parcurge din nou inputul iar fiecare element va fi pus in newArray

pe pozitia egala cu frecventa din arrayFreq,de la pozitia egala cu elementul citit. Ulterior se decrementeaza frecventa si se trece mai departe.Acest algoritm este foarte eficient,avand o complexitate de O(n + k), unde n este numarul de elemente, iar k este intervalul in care se gasesc numerele din input. Daca elementele sunt dintr-un interval al carui maxim superior este foarte mare, vor aparea probleme de efecienta din cauza memoriei(Ex: 10 100 12 10 10 22 -Se aloca un vector cu multe elemente nule) si a informatiei inutile din vectorl arrayFreq(0 reprezinta informatie inuitla).

Daca datele noastre de intrare sunt aproape sortate (ex: 1 2 3 5 4 6 7) se foloseste ShellSort. Este un algoritm asemanator cu InsertionSort, insa este mult mai eficient deoarece elementele nu vor mai fi verificate unul cate unul. Se va alege un gap, ce reprezinta din cat in cat verificam elemente(Spre ex: daca gap = 4 , vom verifica elementele din 4 in 4 incepand de pe pozitia 0.).Practic se vor face comparatii intre elemente de pozitia i, si i+gap.Daca i > i+gap se va face swap pe cele doua elemente.In caz contrar se trece mai departe. Algoritmul se repeta pana cand gap va fi egal cu 0. Initial acesta este egal cu n/2, unde n reprezinta numarul de elemente din vector ce trebuie sortat.La urmatoarea iteratie, gap va fi egal cu previous_gap/ 2, unde previous_gap reprezinta gap de la pasul anterior. Algoritmul ofera o complexitate de O(nlogn) in the best case, iar in the worst case ofera o complexitate de O(n2) (greu de atins deoarece acest algoritm pleaca de la InsertionSort ce are complexitate O(n2) si scade treptat avand in vedere ca gap nu este egal cu 1 ci cu n/2).

Daca datele de intrare difera, fiind pur si simplu un set de numere random, se prefera folosirea unui algoritm hibird, IntroSort. Un algoritm hibribd are la baza mai multi algoritmi ,iar in functie de datele de intrare se aplica un algoritm anume mai intai. Acesta are la baza QuickSort, HeapSort si InsertionSort. Mai intai sortarea este facuta cu QuickSort.Daca recursivitatea depaseste o anumita limita stabilita la intrarea, se trece HeapSort, deoarece se doreste evitarea complexitatii O(n2) a QuikSort pe worst case. Cand numarul de elemente de sortat este foarte mic, se aplica InsertionSort . Apar astfel 3 situatii :

- Daca partitia este mai mare decat limita impus(2*log(n)) IntroSort trece la HeapSort
- Daca partita este mult prea mica, QuicSort trece la InsertionSort
- Daca partitia este sub limita ,dar nu este prea mica atunci se va face QuickSort

1.4 Specificare criteriilor de evaluare alese pentru validarea solutiilor

Pentru cei 3 algoritmi propusi se vor testa urmatoarele : bestCase, midCase si worstCase. Datele vor fi alese pentru a determina un runtime pentru fiecare algoritm.De asemenea, se va testa si acelasi set de date pentru a descoperi care algoritm functioneaza celmai bine pentru un set de date random. Datele de intrare vor avea dimensiuni diferite pentru a putea trasa un grafic al eficientei pentru fiecare algoritm. Cu ajutorul unor grafice, se va putea trage o concluzie in ceea ce priveste algoritmul ideal.

2. Prezentarea Solutiilor

2.1 Descrierea modului in care functioneaza algoritmii alesi

2.1.1 ShellSort

Initial, algoritmul incepe cu un **gap** egal cu jumatate din lungimea vectorului ce trebuie sortat, iar la fiecare pas acesta se reduce cu inca jumatate. Cu noul gap ales, se incepe **InsertionSort**. Primele „**gap - 1**" elemente sunt deja in ordine si se adauga cate un nou element din vector pana cand intreaga sectiune este sortata. Elementul ce trebuie adaugat, este salvat intr-o copie. In vectorul nostru de lungime „**gap**", se cauta pozitia noului element ce trebuie inserat. Odata gasit, se insereaza in pozitia corespunzatoare si se trece la urmatorul gap.

2.1.2 CountingSort

Pentru acest algoritm, mai intai se cauta elementul cu cea mai mare valoare. Odata gasit se creeaza un vector de frecventa de o lungime egala cu **max + 1**. Dupa ce sunt stocate toate aparitiile elementelor, se vor afisa pozitiile vectorului de frecventa ce sunt nenule. Pozitiile vor fi afisate de un numar de ori egal cu valoarea ce se gaseste la indexul respectiv. Astfel, vectorul afisat va fi sortat .

2.1.3 IntroSort

Fiind un algoritm **hibrid**, acesta este construit pe baza a trei algoritmi ce sunt apelati in functie de timpul de executie estimat pentru a sorta vectorul dat ca input. Mai intai, se calculeaza **partitionSize** pentru a stabili cu care algoritm se va incepe sortarea.Exista astfel 3 situatii:

- Daca **partionSize** < 16 se incepe cu **InsertionSort**
- Daca **partitionSize** < 2*log(numarul de elemente ce trebuie sortate) se incepe cu **HeapSort**
- Daca **partitionSize** nu respecta conditiile de mai sus, se incepe cu **QuickSort**

Daca se constata ca sortarea dureaza prea mult, se va schimba algoritmul ales initial.

2.2 Analiza complexitatii solutiilor

Algoritmi au urmatoarele complexitati:

- *ShellSort* : -O(n), unde n reprezinta numarul de elemente(WorstCase)
  -O(n*log(n)), unde n reprezinta numarul de elemente(BestCase)
- *CountingSort* : O(k + n), unde k este cel mai mare element din input, iar n este numarul de elemente a inoutului(WorstCase , BestCase)
- *IntroSort* : O(nLog(n)), unde n reprezinta numarul de elemente(WorstCase,AverageCase)

2.3 Prezentarea principalelor avantaje si dezavantaje pentru solutiile luate in considerare

2.3.1 ShellSort

Avantaje:

ShellSort se remarca prin eficienta pentru dimensiuni medii ale elementelor de input. De asemenea, este de 5 ori mai rapid decat **bubbleSort** ,de 2 ori mai rapid decat **InsertionSort** si mai ales, mai rapid decat orice algoritm cu complexitateaO(n^2).De notat faputl ca este un algoritm ce este eficient pentru elemente duplicate.

Dezavantaje:

Este un algoritm mult mai lent decat **MergeSort**,**HeapSort** si **QuickSort**, mai ales daca datele de intrare sunt foarte mari. „WorstCase" se atinge atunci cand vectorul de input este sortat descrescator.

2.3.2 CountingSort

Avantaje:

Cel mai mare avantaj al algoritmului este reprezentat de complexitatea acestuia (O(n+k)), unde n este dimensiunea vectorului ce trebuie sortat, iar k este dimensiunea vectorului de frecventa.Acestea fiind spuse, acest algoritm se muleaza foarte bine pe un set de date cu valori mici (mai mici de 1000), elemente duplicate si elemente care sunt apropiate ca valoare .

Dezavantaje:

Printre dezavantajele algoritmului se numara situatia in care inputul este sortat deja, datele de intrare sunt foarte mari, si nu poate sorta decat numere intregi.Pentru numerele reale, se pot aplica diferite metode(Se pot indexa la final in vector de frecventa.Spre exemplu, primul numar real va avea indexul **maxArray + 2**, al doilea va avea indexul **maxArray + 3**) insa se va pierde complexitatea de O(n + k) a algorimului.

2.3.3 IntroSort

Avantaje:

Cel mai mare avantaj al algoritmului este evitarea cazlui „WorstCase" al algoritmului **QuickSort.**

Dezavantaje:

Fiind alcatuit din alti 3 algoritmi, exista posibilitatea de instabilitate.

2. Prezentarea Solutiilor

3. Evaluarea Solutiilor

3.1 Descrierea modalitatii de construire a setului de teste

Fiecare algoritm va avea urmatoarele fisiere de input pentru a testat cel mai defavorabil si cel mai favorabil caz .

Input1.txt va arata la fel pentru toti algoritmi(Va trata un sir de numere random pentru a evidentia care este cel mai eficient algoritm)

CountingSort:

test2.in - va testa cel mai defavorabil caz al alg(dimensiune mica a datelor);
test3.in - va testa cel mai defavorabil caz al alg(dimensiune mare a datelor);
test4.in - va testa cel mai favorabil caz al alg(dimensiune mica a datelor);
test5.in - va testa cel mai favorabil caz al alg(dimensiune mare a datelor);
test6.in - test random
test7.in - test random
test8.in - va testa un fisier ce va contine doar "1" si "2"
test9.in - va testa un fisier cu un singur element
test8.in - va testa un fisier gol

IntroSort:

test2.in - va testa cel mai defavorabil caz al alg(de fapt va fi format de worstCaseQuickSort, worstCaseHeapSort, worstCaseInsertionSort)
test3.in - va testa cel mai favorabil caz pe un set de date foarte mare;
test4.in - va testa cel mai favoravil caz pe un set de date foarte mic;
test5.in - va testa un fisier gol("Skip the file")
test6.in - va testa un fisier ce va contine doar "1" si "2"
test7.in - va testa 8000 de elemente
test8.in - va testa 1 element
test9.in - test random
test10.in - test random

ShellSort:

test2.in - va testa cel mai ineficient caz
test3.in - va testa cel mai eficient caz
test4.in - test random
test5.in - test random
test6.in - va testa un fisier ce va contine "1" si "2"
test7.in - va testa un set de date mic
test8.in - va testa un fisier gol
test9.in - va testa un set de date mic
test10.in - va testa un set de date mic

3.2 Specificatiile Sistemului de calcul

Testele au fost rulate pe un Intel i7 Core vPro, 8 GB RAM.

3.3 Graficele ce ilustreaza rezultatele evaluarii solutiilor

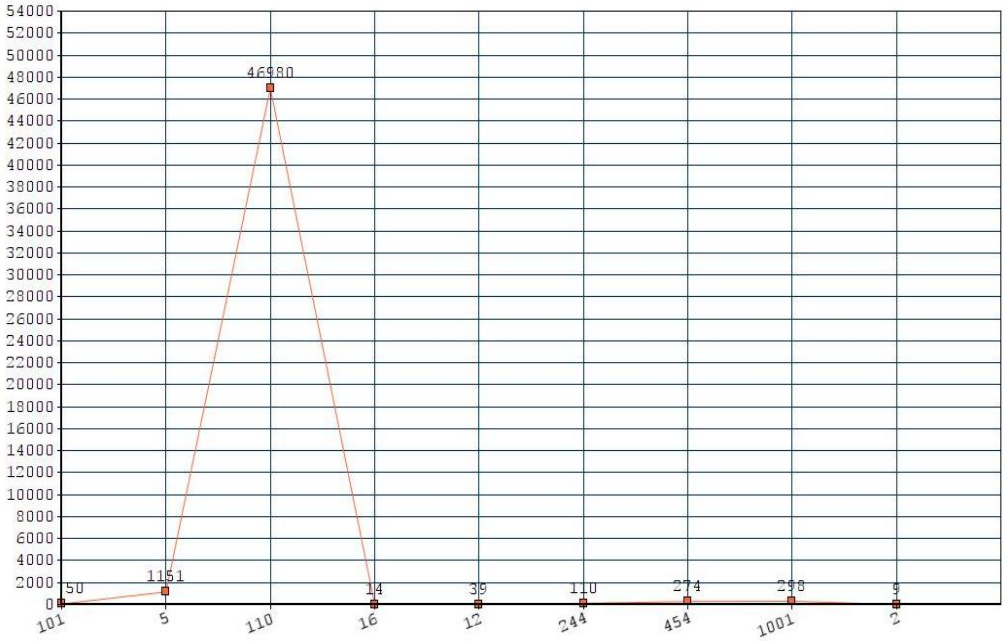

Fig1. CountingSort grafic timp[milisecunde] / dimensiune date

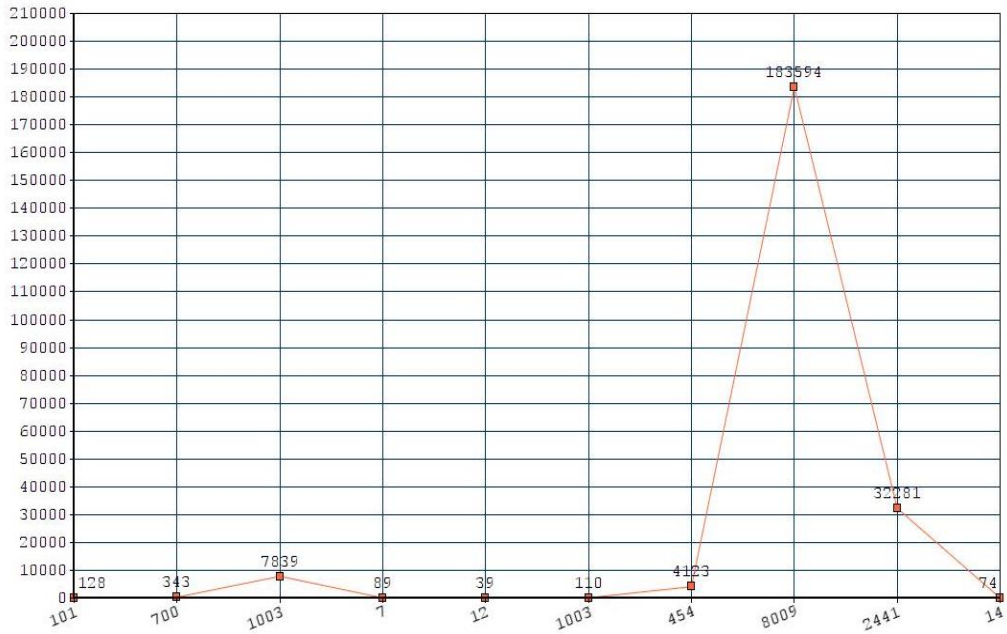

Fig2. IntroSort grafic timp[milisecunde] / dimensiune date

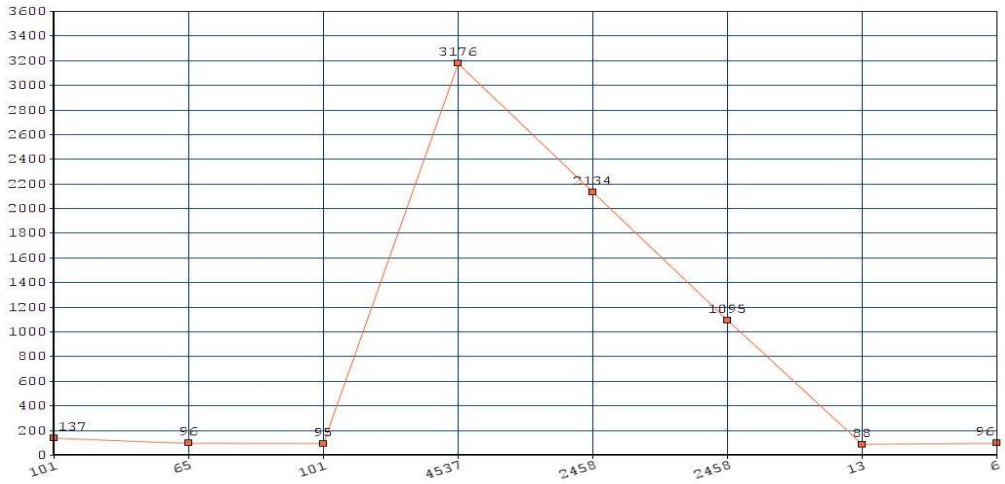

Fig3. ShellSort grafic timp[milisecunde] / dimensiune date

3.4 Prezentarea, succinta, a valorilor obtinute pe teste

Fiecare algoritm in parte va afisa, vectorul de numere primit ca input, sortat.

4. Concluzie:

Fiecare algoritm are performate diferite in situatii diferite. Astfel nu putem stabili care este cel mai bun algoritm, insa vom sti cu certitudine care are cea mai buna performanta pe un caz particular

Referinte:

1. https://www.geeksforgeeks.org/shellsort/
2. https://www.geeksforgeeks.org/counting-sort/
3. https://www.geeksforgeeks.org/know-your-sorting-algorithm-set-2-introsort-cs-sorting-weapon/
4. https://unacademy.com/lesson/shell-sorting-algorithm-advantages-and-disadvantages/CQSB6V60