Tema Analiza Algoritmilor

Problema aleasa– Sortare

Algoritmi utilizati- QuickSort, MergeSort, HeapSort

Margineanu Nicolae-Vladut

Grupa 313CA, Facultatea de Automatica si Calculatoare

Universitatea Politehnica Bucuresti

vladut.margineanu@gmail.com

15 noiembrie 2018

Abstract Tema are ca obiectiv initierea si dezvoltarea cunostintelor in analizarea unei probleme practice avand ca solutii algoritmi specializati din punct de vedere spatial si temporal. Fiecare metoda de rezolvare este insotita de explicatii referitoare la implementarea acesteia, de analiza complexitatii si de cazurile favorabile, medii si defavorabile in functie de spatiu si timp.

1 Introducere

Tema are la baza rezolvarea problemei de sortare impreuna cu un studiu comparativ privind principalii algoritmi care rezolva aceasta problema (QuickSort, MergeSort, HeapSort). Aceasta cuprinde descrierea problemei, specificarea solutiilor alese si criteriile de evaluare pentru solutia propusa. Studiul este realizat pe seturi de date cat mai variate pentru a evidentia principalele criterii de alegere (avantaje si dezavantaje) pentru algoritmii analizat.

1.1 Descrierea problemei rezolvate

Mentionarea unei aplicatii practice: sortarea datelor dintr-o zona sau dintr-un tabel din Excel (Office). Sortarea datelor este o parte integranta a analizei datelor. Este posibil sa dorim sa punem o lista de nume în ordine alfabetica, sa compilam o lista de niveluri ale unui inventar de produse de la cel mai mic la cel mai mare nivel.

1.2 Exemple de aplicatii practice pentru problema aleasa

Sortarea datelor ne permit vizualizarea rapida a datelor si o mai bună intelegere a acestora, organizarea si gasirea datelor dorite. Putem sa sortam datele dupa text (de la A la Z sau de la Z la A), după numere (de la cel mai mic la cel mai mare sau invers) și după date și ore (de la cea mai veche la cea mai nouă sau invers) în una sau mai multe coloane. De asemenea, putem sa sortam dupa o lista particularizata pe care o cream (cum ar fi Mare, Mediu sau Mic) sau dupa format, inclusiv culoarea celulei și a fontului sau setul de pictograme.

1.3 Specificarea solutiilor alese cu detalii despre implementare

Pentru rezolvarea problemei, am ales urmatoarele solutii:

A) QuickSort- este un algoritm de tip divide et impera. Acesta divide mai intai un vector in doi vectori mai mici: elementele mici si elementele mari. Dupa care acesta sorteaza recursiv subvectorii. Pasii sunt urmatorii:

- Alege un element numit pivot din vector.
- Partitionarea: reordoneaza vectorul astfel incat toate elementele cu valori mai mici decat pivotul sa ajunga inaintea pivotului, in timp ce toate elementele cu valori mai mari decat pivotul vin dupa el (valorile egale merg in orice parte). Dupa aceasta partitionare, pivotul este in pozitia finala (se numeste operatia de partitionare).
- Se aplica recursivitatea subvectorului cu elementele cu valori mai mici si separat, subvectorului cu valori mai mari.

B) MergeSort- este un algoritm de tip divide et impera si are la baza comparatia. Conceptual, algoritmul urmeaza pasii:

- Imparte lista nesortata in n subliste, fiecare continand cate un element (o lista cu un element este considerata sortata).
- In mod repetat, imbinam sublistele pentru a realiza noi subliste sortate pana cand vom avea doar o sublista ramasa. Aceasta va fi lista sortata.

C) HeapSort- este un algoritm bazat pe comparatie. Acesta poate fi impartit in doua parti. In prima etapa, este construita o structura de date a unui arbore binar (de exemplu printr-un vector). Stim ca pozitia nodului radăcina in vector este 0, iar pentru fiecare nod in parte, parintele si descendentii se pot calcula dupa formulele:

Parinte(i) = (i - 1) / 2, unde i este indicele nodului curent.

IndexStanga(i) = 2 * i + 1, unde i este indicele nodului curent.

IndexDreapta(i) = 2 * i + 2, unde i este indicele nodului curent.

In a doua etapa, vectorul sortat este creat prin eliminarea repetata a celui mai mare element din heap (radacina heap-ului), si inserarea acestuia in vector. Heap-ul este actualizat dupa fiecare extragere pentru a mentine propietatea de heap. Odata ce toate elementele au fost extrase din heap, rezultatul este un vector sortat.

1.4 Specificarea criteriilor de evaluare alese pentru validarea solutiilor

O modalitate de validare a corectitudinii pentru QuickSort este urmatoarea: verificarea setului de teste (date ca input acestui algoritm) pe solutiile MergeSort, HeapSort si Bubble Sort. Prin acest mod, vom observa corectitudinea solutiei, complexitatea temporala si spatiala (cel mai defavorabil, cel mai favorabil si mediu comportament), memoria folosita, stabilitatea.

QuickSort foloseste metoda partitionarii, MergeSort, metoda imbinarii, iar HeapSort, metoda selectiei. Bubble Sort foloseste metoda interschimbarii. Prin verificarea fiecarui algoritm pe acelasi set de date, vom constata eficienta fiecaruia, avantajele si dezavantajele fiecaruia in practica, prin comparatie. Input-urile vor fi formate din numere intregi (date in ordine crescatoare, descrescatoare si aleatorie pentru

verificarea complexitatii), din stringuri formate din caractere (A-Z si a-z) si o lista particularizata (cum ar fi Mare, Mediu sau Mic). Am creat diferite inputuri realizate cu generatoarele incluse in arhiva de la etapa 2. Aceste generatoare realizeaza numere in ordine aleatoare si siruri de caractere formate din toate caracterele pe care se pot face sortare in ordine ascendenta.

In concluzie, principala metoda de evaluare este verificarea solutiei respective prin testarea altor algoritmi de sortare, cum ar fi QuickSort, MergeSort, HeapSort la diferitele tipuri de input-uri.

2 Prezentarea solutiilor

2.1 Descrierea modului in care functioneaza algoritmii alesi

A) QuickSort- pasii algoritmului sunt urmatorii:

1) Se alege un element al listei, denumit pivot

2) Se reordoneaza lista astfel incat toate elementele mai mici decat pivotul sa fie plasate inaintea pivotului si toate elementele mai mari sa fie dupa pivot. Dupa aceasta partitionare, pivotul se afla in pozitia sa finala.

3) Se sorteaza recursiv sublista de elemente mai mici decat pivotul si sublista de elemente mai mari decat pivotul.

Observatie: o lista de dimensiune 0 sau 1 este considerata sortata.

Quicksort efectuează sortarea bazandu-se pe o strategie divide et impera. Astfel, el imparte lista de sortat in doua subliste mai usor de sortat.

Pseudocod QuickSort:

function QUICKSORT(A, inf, sup) is

| i← inf

| j← sup

| x← A[(i+j) div 2]

| **repeat**

| | **while** (i<sup)/\ (A[i]<x) **execute** i← i+1

| | **while** (j>inf) /\ (A[j]>x) **execute** j← j-1

| | **if** i <=j **then**

| | |t← A[i]; A[i]← A[j]; A(j)← t

| | |i← i+1; j← j-1

| | |___■

| **until** (i>j)| **if** (inf<j) **then** QUICKSORT(A, inf, j)

| **if** (i<sup) **then**QUICKSORT(A, i, sup)

|___■

Alegerea pivotului cu regula “Median-of-threee" se realizeaza astfel:

```
mid := (lo + hi) / 2
if A[mid] < A[lo]
    swap A[lo] with A[mid]
if A[hi] < A[lo]
    swap A[lo] with A[hi]
if A[mid] < A[hi]
    swap A[mid] with A[hi]
pivot := A[hi]
```

Această regulă "mediană de trei" contracarează intrarea sortată (sau sortată invers) și oferă o estimare mai bună a pivotului optim (median adevărat) decât selectarea unui singur element, atunci când nu există informații despre ordonare si este cunoscută intrarea.

B) MergeSort- algoritmul executa urmatorii pasi:

1) Dacă lista este de lungime 0 sau 1, atunci este deja sortată. Altfel:

2) Împarte lista nesortată în două subliste aproximativ egale.

3) Sortează fiecare sublistă recursiv prin reaplicarea algoritmului merge sort.

4) Se interclasează cele două liste și se obține lista inițială sortată.

Este un exemplu de algoritm de tip divide et impera.

Pseudocod MergeSort:

```
function mergesort(m)
    var list left, right
    if length(m) ≤ 1
        return m
    else
        middle = length(m) / 2
    for each x in m up to middle
        add x to left
    for each x in m after middle
        add x to right
    left = mergesort(left)
```

```
        right = mergesort(right)
        result = merge(left, right)
    return result
end function
```

Functia merge(), pseudocod:

```
function merge(left,right)
    var list result
    while length(left) > 0 and length(right) > 0
         if first(left) ≤ first(right)
             append first(left) to result
             left = rest(left)
         else
             append first(right) to result
             right = rest(right)
    if length(left) > 0
        append left to result
    if length(right) > 0
        append right to result
    return result
end function
```

Exista mai multe variante pentru functia de imbinare merge(), dar am incercat sa realizez cea mai simpla varianta.

C) HeapSort- pasi pentru o varianta de implementare a algoritmului HeapSort:

1) Presupunem ca vectorul formeaza un arbore binar, fiecare pozitie din vector reprezentand un nod, cu radacina pe pozitia 0 (zero) si cu fiecare nod k avand copiii 2k+1 si 2k+2 (daca nu exista pozitia din vector cu indicele respectiv, atunci nu exista nod copil ⇒ NULL)

2) Formam un max-heap cu aceeasi reprezentare (pe vector, fara a construi alta structura pentru noduri)

3) Extragem maximul din radacina heap-ului (pozitia 0 din vector) si facem o interschimbare intre pozitia maximului si ultima pozitie din vector. Acum maximul se afla pe pozitia dorita si putem sa-l excludem din heap.

4) Repetam pasii (refacem forma de heap, extragem noul maxim, reducem cu 1 numarul de elemente nesortate), cat timp mai sunt elemente in heap.

Pseudocodul pentru HeapSort este:

```
Heapsort(A as array)
   BuildHeap(A)
   for i = n to 1
      swap(A[1], A[i])
      n = n – 1
      Heapify(A, 1)
BuildHeap(A as array)
   n = elements_in(A)
   for i = floor(n/2) to 1
      Heapify(A,i,n)Heapify(A as array, i as int, n as int)
   left = 2i
   right = 2i+1
   if (left <=  n) and (A[left] > A[i])
      max = left
   else
       max = i
  if (right<=n) and (A[right] > A[max])
      max = right    if (max !=  i)
      swap(A[i], A[max])
      Heapify(A, max)
```

HeapSort este un algoritm eficient de sortare implementat cu structura de date Heap.

2.2 Analiza complexitatii solutiilor

A) QuickSort

Cazul cel mai favorabil- O(n*log n)

In cazul cel mai echilibrat, de fiecare data cand efectuam o partitie divizam lista în doua bucati aproape egale. Aceasta inseamna ca fiecare apel recursiv proceseaza o lista cu jumatate din dimensiune. In consecinta, inainte de a ajunge la o lista cu marimea 1, putem face numai apeluri log2 n imbricate. Aceasta inseamna ca adancimea arborelui de apel este log2 n. Dar nici doua apeluri la acelasi nivel al arborelui de apel nu procesează aceeasi parte a listei originale; astfel incat fiecare nivel al apelurilor are nevoie numai de timpul O(n) toate impreuna (fiecare apel are nişte constante, dar exista doar apeluri O(n) la fiecare nivel, acesta este subsumat in factorul O(n). Rezultatul este ca algoritmul foloseste doar timpul O(n*log n).

Cazul mediu- O(n*log n)

Cazul cel mai defavorabil- O(n^2)

Partea cea mai dezechilibrata apare atunci cand una dintre sublistele returnate de rutina de partitionare are dimensiunea n - 1. Acest lucru se poate întâmpla daca pivotul se întampla sa fie cel mai mic sau cel mai mare element din lista sau in unele implementari atunci cand toate elementele sunt egale.

Daca se intampla acest lucru in mod repetat in fiecare partitie, fiecare apel recursiv proceseaza o lista cu o dimensiune mai mica decat lista precedenta. In consecinta, putem face n - 1 apeluri imbricate inainte de a ajunge la o lista de dimensiune 1. Aceasta inseamna ca arborele de apel este un lant liniar de n - 1 apeluri imbricate. Ajungem la a face O (n - i) pentru a face partitia si n + (n - 1) + (n - 2) + …. + (n – n) = O (n^2) , deci in acest caz Quicksort ia O (n^2) timp.

B) MergeSort

Cazul cel mai favorabil = Cazul mediu = Cazul cel mai defavorabil = O(n*log n)

In sortarea a n obiecte, MergeSort are aceeasi complexitate pentru cele trei cazuri, O(n*log n). Daca timpul de rulare de sortare pentru o lista cu lungimea n este T(n), atunci recurenta T(n) = 2*T (n/2) + n rezulta din definitia algoritmului (se aplica algoritmul pe doua liste de dimensiune jumatate din dimensiunea listei originale si se adauga pasii n luati pentru a imbina cele două liste care rezulta). Forma finala rezulta din “master theorem for divide-and-conquer recurresces”.

In cazul cel mai defavorabil, numarul de comparatii pe care MergeSort il face este dat de numerele sortate. Aceste numere sunt egale sau putin mai mici decat (n * lg n –2 ^ (lg n) + 1) care este intre numerele (n lg n - n + 1) si (n lg n + n + O(lg n)).

C) HeapSort

Cazul cel mai favorabil- O(n)

Daca toate cheile sunt diferite, atunci complexitatea este O(n * log n)

Cazul mediu- O(n * log n)

Cazul cel mai defavorabil- O(n * log n)

Inaltimea unui arbore binar complet care contine n elemente este log (n).

Operatia de heapify ia O(log n) timp. Atunci cand schimb nodul max / min cu altul din baza (cel mai de jos) al heap-ului, va trebui sa dau “push” nodului inapoi in partea de jos. Deoarece exista n elemente si

inaltimea heap-ului este egala cu log (n), voi face log (n) swap-uri pe masura ce nodul meu traverseaza heapul in jos. Daca repet acest proces de n ori, timpul va fi O(n*log n).

2.3 Prezentarea principalelor avantaje si dezavantaje pentru solutiile luate in considerare

A) QuickSort

Avantaje: Complexitatea sa de timp medie pentru a sorta un vector de n elemente este O(n*log n).

In medie, ruleaza foarte repede, chiar mai rapid decat Merge Sort.

Nu necesita memorie suplimentara.

Dezavantaje: Timpul de rulare poate diferi in functie de continutul vectorului.

Timpul de rulare al lui Quicksort se degradeaza dacă se da un vector care este aproape sortat (sau chiar descrescator sortata). Cel mai defavorabil caz de rulare, O (n^2) pentru a sorta o serie de elemente n, se intampla cand este data o matrice sortata.

Nu este stabila.

B) MergeSort

Avantaje: Solutia este intotdeauna rapida pentru structurile mari de date.

Chiar si in cel mai defavorabil caz, timpul de executie este O(n*log n).

Este o solutie stabila.

Dezavantaje: Algoritmul poate folosi foarte multa memorie.

Utilizeaza spatiu suplimentar proportional cu n. Acest lucru poate incetini cand se sorteaza date foarte mari.

Spatiu auxiliar: Mergesort foloseste un spatiu suplimentar, QuickSort necesita puțin spatiu. QuickSort este un algoritm de sortare pe loc. Asta inseamna ca nu necesita spatiu suplimentar de stocare pentru a efectua sortarea. MergeSort necesita un vector temporar pentru a imbina vectorii sortati.

C) HeapSort

Avantaje: Eficienta- algoritmul este eficient. Performanta este optima. Aceasta implica faptul ca nici un alt algoritm de sortare nu poate funcționa mai bine in comparatie cu acesta.

Utilizarea memoriei este mai mica- utilizarea memoriei este minimă deoarece, in afara de ceea ce este necesar pentru a pastra lista initiala de elemente care urmeaza sa fie sortate, nu are nevoie de spatiu suplimentar de memorie pentru a functiona. In schimb, algoritmul de sortare MergeSort necesita mai mult spatiu de memorie. In mod similar, algoritmul QuickSort necesita mai mult spatiu pe stiva datorita naturii sale recursive.

Consistenta- algoritmul de sortare Heap prezintă performanță consistentă. Acest lucru înseamnă că acesta se comportă la fel de bine în cele mai bune, medii și cele mai grave cazuri. Datorită performanței sale garantate, este deosebit de potrivit să se utilizeze în sisteme cu timp de răspuns critic.

Dezavantaje: Este un algoritm instabil. Un algoritm de sortare este stabil daca menține ordinea relativa a elementelor care au aceeași cheie (adică modul in care acestea sunt prezente in vectorul initial). HeapSort poate rearanja ordinea relativa.

Factorii constanti- in implementarea de zi cu zi, exista factori constanti pe care analiza teoretica nu le ia in considerare. In cazul lui Heapsort vs. Quicksort, se constata ca exista modalitati (de exemplu, mediana de trei) pentru a face ca cazurile cele mai defavorabile sa fie rare ale Quicksort-ului. Avand un vector cu o distributie normala, Quicksort si Heapsort vor rula ambele in O (n log (n)). Dar Quicksort se va executa mai repede, deoarece factorii constanti sunt mai mici decat factorii constanti pentru Heapsort. Partitionarea este mai rapida decat mentinerea heap-ului (care consuma mai multe resurse).

3 Evaluare

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

Setul de teste este format din elemente de tip int, realizate cu ajutorul programului generator_number_int.cpp implementat de mine. Testele sunt ordonate in functie de complexitate si memorie folosita (primul are doar 10 elemente din intervalul [0,100], iar ultimul are 10000 elemente din intervalul [-1000,10000], avand si elemente negative).

Testele suplimentare sunt organizate astfel:

- primul contine elemente de tip float. Acesta a fost generat cu programul generator_number_float.cpp.

- al doilea contine siruri de caractere cuprinse intre a-z. Acestea au fost generate cu programul generator_string_a_z.cpp

-al treilea contine siruri de caractere cuprinse intre a-z, A-Z, 1-2 (avand caractere din intervalele date). Acestea au fost generate cu programul generator_stringAa_Zz.cpp.

Toate programele generatoare de teste suplimentare au fost implementate de mine.

Testele de referinta au fost realizate cu algoritmul de sortare Bubble Sort din STL. In urma realizarii acestora, am comparat rezultatele date de solutiile mele cu cele obtinute cu Bubble Sort, si am verificat corectitudinea acestora.

3.2 Mentionarea specificatiilor sistemului de calcul pe care am rulat testele

Procesor

Model name: Intel(R) Core(TM) i3-3110M CPU @ 2.40GHz

Memorie disponibila

Total online memory: 4G

3.3 Ilustrarea rezultatelor evaluarii solutiilor pe setul de teste

Voi prezenta in continuare rezultate exacte pentru cazul mediu obtinute pe urmatorele tipuri de algoritmi de sortari: QuickSort, MergeSort, HeapSort, InsertionSort. De exmplu (Fig 1):

Quicksort: 11.667(n+1)ln(n)−1.74n−18.7411.667(n+1)ln⁡(n)−1.74n−18.74

Mergesort: 12.5nln(n)12.5nln⁡(n)

Heapsort: 16nln(n)+0.01n16nln⁡(n)+0.01n

Insertionsort: 2.25n2+7.75n−3ln(n)2.25n2+7.75n−3ln(n)

m

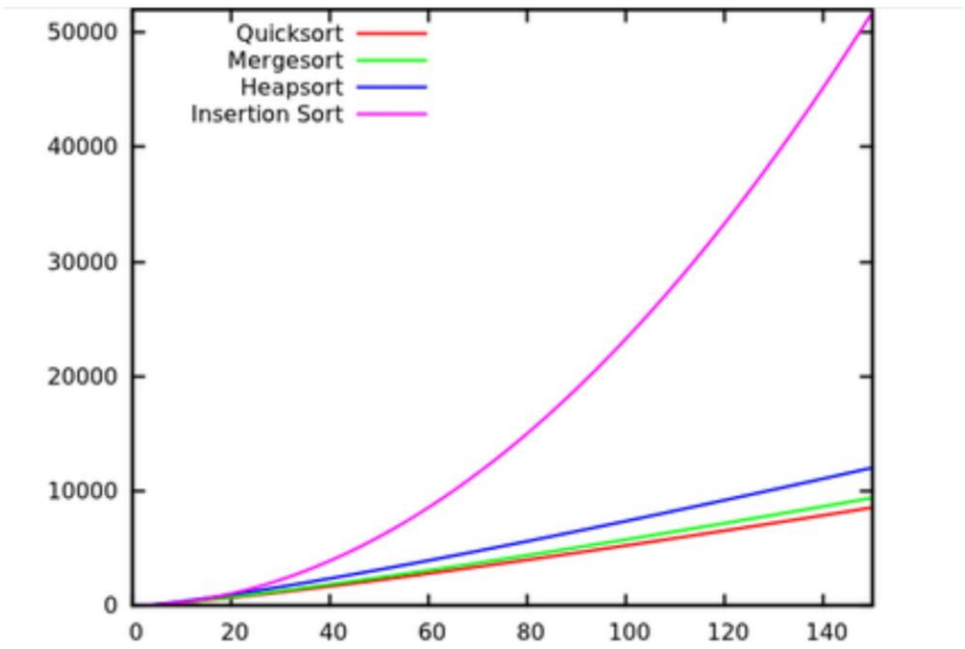

Fig 1: Ilustrarea rezultatelor evaluarii solutiilor pe un set de teste.

Aceste rezultate indica faptul ca Quicksort este cel mai rapid. In continuare, vom observa ca algoritmii se comporta diferit la input-uri mici:

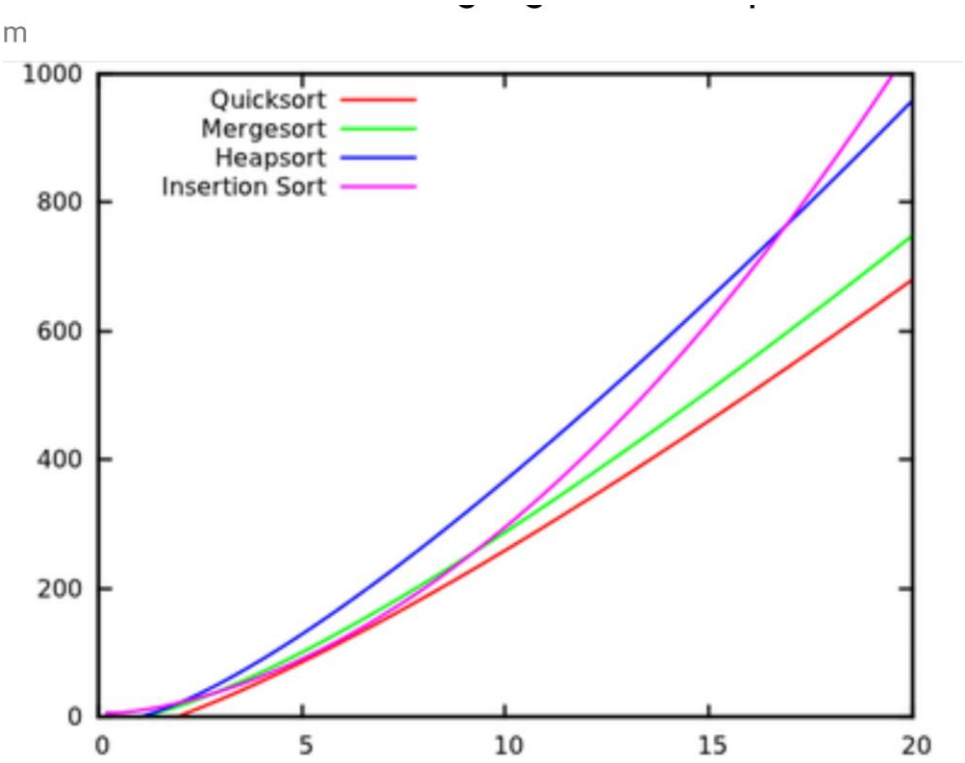

Fig 2: Ilustrarea rezultatelor evaluarii solutiilor pe un test cu input mic.

3.4 Prezentarea valorilor obtinute pe teste

Valorile obtinute pe testele date, dupa cum ma asteptam, au fost ordonate in ordine ascendenta atat numerele de tip int, de tip float, cat si stringurile formate din caractere diferite.

4 Concluzii

In practica, principalele caracteristici de care trebuie sa tinem cont in alegerea solutiei este:

A) QuickSort- ruleaza foarte repede, chiar mai rapid decat Merge Sort.

B) MergeSort- foloseste un spatiu suplimentar. Acesta necesita un vector temporar pentru a imbina vectorii sortati.

C) HeapSort- algoritmul de sortare Heap prezintă performanță consistentă. Acest lucru înseamnă că acesta se comportă la fel de bine în cele mai bune, medii și cele mai grave cazuri.

In concluzie, QuickSort poate fi ales in practica deoarece ruleaza foarte repede, chiar mai rapid decat Merge Sort si nu necesita memorie suplimentara (solutia optima).

Bibliografie

1. Knuth, Donald (1998). "Section 5.2.4: Sorting by Merging". *Sorting and Searching*. The Art of Computer Programming. 3 (2nd ed.).

2. Demuth, H. Electronic Data Sorting. PhD thesis, Stanford University, 1956.

3. Sedgewick, Robert (1 September 1998). Algorithms In C: Fundamentals, Data Structures. Sorting, Searching, Parts 1-4 (3 ed.). Pearson Education.

4. Sedgewick, R. (1978). "Implementing Quicksort programs".

5. Skiena, Steven (2008). "Searching and Sorting". The Algorithm Design Manual. Springer.