Colegiul National “Vasile Alecsandri”
Galati

Studiu asupra timpilor de sortare

Paunoiu Alexandru Dumitru
Clasa a X-a

2007-2008

Descrierea lucrarii

Aceasta lucrare este dedicata studiului timpilor de sortare, si imbunatatirii acestora prin propunerea de solutii noi (solutiile propuse sunt in scopul micsorarii timpului de executie si mai putin in scopul micsorarii memoriei utilizate) sau prin extinderea algoritmilor existenti. Accentul va cadea asupra Quicksort-ului si Bubblesort-ului.

Toate implementarile vor fi realizate in C/C++.

Cuprins

1. Introducere 4
    - 1.1. Elemente fundamentale 4
    - 1.2. Exemple 4
2. SEP (Sort Elements as Positions) 6
    - 2.1. Descriere 6
    - 2.2. Implementare 6
    - 2.3. Avantaje si dezavantaje 7
    - 2.4. Extinderea SEP-ului cu Quicksort 8
        - 2.4.1. Descriere 8
        - 2.4.2. Algoritmul 8
        - 2.4.3. Implementare 8
        - 2.4.4. Analiza complexitatii si a consumului de memorie 10
        - 2.4.5. Avantaje si dezavantaje 10

Rezumat 11

Bibliografie 12

1. Introducere

1.1. Elemente fundamentale

Definitie: Prin sortare intelegem algoritmul prin care putem rearanja k elemente intr-o anumita ordine (de exemplu: in ordine lexicografica, ordine crescatoare).

Sortarea este des folosita in lucrul cu liste. Un exemplu de folosire a sortarii il reprezinta motoarele de cautare web, care folosesc astfel de algoritmi (Google, Yahoo, MSN).

Exista diversi algoritmi de sortare printre care se numara: Quicksort, Heapsort, Bubblesort, fiecare avand avantaje si dezavantaje. Fiecare algoritm de sortare se bazeaza pe o anumita metoda: partitionare (Quicksort), comparare (Bubblesort).

Dupa cum bine se stie, fiecare algoritm are nevoie de un anumit spatiu de memorie, si fiecare realizeaza sortarea intr-un anumit timp. Pentru a masura timpul de executie vom folosi notiunea de complexitate (O(*x*), unde *x* reprezinta numarul de comparatii facute de algoritm si citim “complexitate *x*”).

De asemenea, vom folosi notatia *O* pentru a defini memoria folosita de algoritm.

Inainte de a se explica cum se masoara *x*, trebuie clarificate cateva notiuni. Prin *caz defavorabil* intelegem cazul in care timpul de executie va fi cel mai mare si poate fi luat ca o margine superioara. *Cazul mediu* il folosim atunci cand datele de intrare nu sunt clare, deci, toate sunt la fel de probabile. Se va folosi log *x* si astfel se va intelege log in baza 2 din *x*, iar in alte cazuri, daca baza nu este 2, se va preciza cat este aceasta.

Depinzand de algoritm, *x* va fi masurat pentru cel mai defavorabil caz. Daca exista un caz mediu, *x* va fi masurat pentru acesta. Spre exemplu: Quicksort are O(n log n), dar in cel mai defavorabil caz are $O(n^2)$, Bubblesort nu admite caz mediu si intotdeauna va avea $O(n^2)$.

1.2. Exemple

Bubblesort are timpul de executie $O(n^2)$, iar memoria folosita este O(1).
Avantaje: implementare usor de realizat, algoritm usor de retinut si de inteles
Dezavantaje: timpul de executie

Implementare:

```
/*
implementarea exemplifica sortarea vectorului x in ordine
crescatoare
n: lungimea vectorului x
swap: variabila auxiliara care verifica daca mai sunt elemente
in ordine descrescatoare
*/
do{
      swap=0;
```

```
        for(i=0;i<n;i++)
              if(x[i]>x[i+1])
              {
                    swap(x[i],x[i+1])
                    swap=1;
              }
  }while(swap)
```

Quicksort are complexitatea timpului de executie O(n log n) in cazul mediu, acesta fiind folosit in mod obisnuit pentru a-l defini, dar in cel mai defavorabil caz are $O(n^2)$. Memoria folosita este O(log n). Exista implementat in libraria stdlib.h, dar prezinta bug-uri asa ca se recomanda implementarea lui “de mana”.

Avantaje: timpul de executie

Dezavantaje: memoria folosita, intelegerea si implementarea algoritmului.

Implementare:

```
/*
implementarea exemplifica sortarea vectorului x in ordine crescatoare
se foloseste partitia Lomuto pentru o implementare mai usoara
utilizare: quicksort(indice_inceput,indice_final);
*/
void quicksort(int l,int u)
{
    int i,m;
    if(l>=u) return;
    m=l;
    for(i=l+1;i<=u;i++)
        if(x[l]>x[i])
        {
            m++;
            swap(x[i],x[m]);
        }
    swap(x[l],x[m]);
    quicksort(l,m-1);
    quicksort(m+1,u);
}
```

Heapsort, alt algoritm de sortare, este poate cel mai eficient (desi nu va fi analizat in aceasta lucrare) ca timp si memorie folosita (complexitate este O(n log n), iar memoria folosita este O(1)). Este gata implementat in libraria STL.

Avantaje: timpul de executie, memoria folosita, implementarea (se poate utiliza STL).

2. SEP (Sort Elements as Positions)

2.1. Descriere

SEP este o noua abordare asupra sortarii, venind, astfel, cu o noua metoda. Ideea a pornit de la calcularea frecventei de aparitie a unui element intr-un anumit vector.

Aceasta sortare are ca principiu de lucru stocarea fiecarui element *i* ce trebuie sortat ca pozitie al unui alt element dintr-un vector, astfel: v[a[i]]++ (se incrementeaza pentru a verifica de cate ori apare acel element), unde *a* reprezinta vectorul ce trebuie sortat (vezi figura 1). Metoda nu este una clasica.

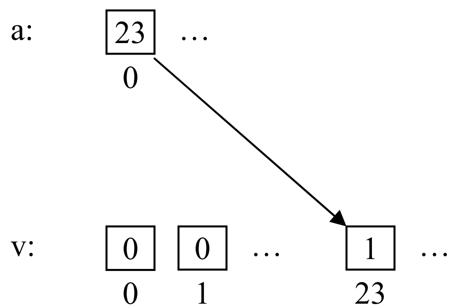

figura 1

Algoritmul are complexitatea O(n), deoarece vectorul *a* este parcurs o singura data.

Memoria reprezinta un dezavantaj, deoarece numarul de elemente al vectorului sortat trebuie sa fie max(a[i]), asadar memoria are complexitatea O(max(a[i])).

2.2. Implementare

```
/*
n: dimensiunea vectorului ce trebuie sortat
*/
void sep()
{
    int i;
    for (i = 0; i < n; i++)
        v[a[i]]++;
}
```

Dupa cum se poate observa atat implementarea, cat si timpul de executie reprezinta un avantaj. Problema timpului revine insa in momentul in care dorim sa afisam vectorul sortat. Pentru aceasta vom mai adauga cateva instructiuni algoritmului pentru a face afisarea cat mai optima.

```c
/*
n: dimensiunea vectorului ce trebuie sortat
min: valoarea de la care se va porni afisarea
max: valoarea pana la care se va afisa
*/
void sep()
{
    int i;
    min = max = a[0];
    for (i = 0; i < n; i++)
    {
        if (min > a[i]) min = a[i];
        if (max < a[i]) max = a[i];
        v[a[i]]++;
    }
}
```

In acest fel afisarea va porni de la min si se va sfarsi la max, parcurgand astfel toate elementele din intervalul (min,max), iar daca elementul exista atunci se va afisa.

Complexitatea algoritmului de afisare va fi: O(n+max-min+1), *max-min+1* fiind lungimea intervalului (min,max). Implementarea afisarii este:

```c
void afisare()
{
    int i,j;
    for (i = min; i <= max; i++)
    {
        for(j = 1; j <= v[i]; j++)
            printf("%d ",i); //i reprezinta elementul k din a
    }
}
```

Afisarea este destul de greoaie daca *max-min+1* este mare.

2.3. Avantaje si dezavantaje

Acest algoritm nu este perfect, suferind la capitolul viteza de afisare si memorie alocata, dar poate reprezenta un avantaj enorm cand este vorba de viteza algoritmului in sine. In continuare se vor puncta avantajele si dezavantajele acestuia.

Avantaje: viteza algoritmului, elementele putand fi sortate odata cu citirea lor, viteza de afisare (doar daca intervalul (min, max) este mic sau aproximativ toate elementele din acesta exista), usurinta cu care se poate implementa si intelege.

Dezavantaje: viteza afisarii (daca intervalul (min,max) este mare), memoria folosita (*O(max(a[i]))*), se poate folosi doar pentru sortari numerice.

Putem spune din privinta timpului de executie ca: SEP < Quicksort < Bubblesort, iar din privinta memoriei ca Bubblesort < Quicksort < SEP.

2.4. Extinderea SEP-ului cu Quicksort

2.4.1. Descriere

Acest pas este unul inevitabil, deoarece SEP consuma prea multa memorie, memorie de care nu putem dispune in anumite cazuri. Totodata, prin rezolvarea acestui nou algoritm, putem sorta atat numeric, cat si lexicografic.

Cu alte cuvinte, se va implementa un algoritm hibrid, cu un timp de executie mai mic decat cel al Quicksort-ului, si cu un consum de memorie mic *comparativ* cu SEP.

2.4.2. Algoritmul

In urmatoarele randuri se va descrie algoritmul, ca mai tarziu sa fie prezentata implementarea si apoi calculata complexitatea si memoria consumata de acesta.

Vom considera o sortare numerica in ordine crescatoare. Imaginati-va ca vrem sa incadram fiecare numar in functie de o regula si anume numarul de cifre. Pentru fiecare categorie numerotata de la 1 la 9 (tipul long are maximum 9 cifre), vom avea mai multe numere (vezi figura 2). Cele *m* numere dintr-o categorie trebuie sa fie si ele sortate, aici intervenind Quicksort-ul.

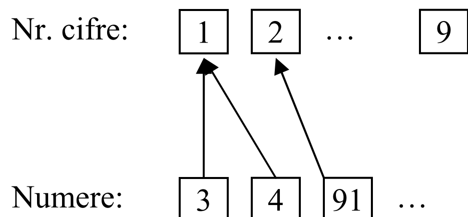

figura 2

Pentru a intelege mai bine se va da un exemplu: la o agentie de modelling trebuie "sortate" *n* manechine, cea mai neindicata avand cea mai mica inaltime, iar cea mai indicata avand cea mai mare inaltime. Dupa ce am ales fiecarei inaltimi un manechin, se poate intampla ca mai multe manechine sa aiba aceeasi inaltime. Atunci, pentru fiecare inaltime, vom "sorta" manechinele in functie de cat de frumoase sunt.

Exista doua criterii: numarul de cifre ale unui numar si marimea lor pentru un anumit numar de cifre *i*.

2.4.3. Implementare

Daca algoritmul de mai sus s-a inteles, putem trece la implementarea acestuia. In comentariile din implementare se va pastra exemplul cu manechinele.

```
        struct structura{
            int categorie[51];
            int lungime;
        }vector[10]; //fiecare element din vectorul de tip structura
                     //va reprezenta primul criteriu (inaltimea), iar
                     //pentru fiecare criteriu avem un vector
                     //categorie unde se retine al doilea criteriu
                     //(frumusetea)

        //vom utiliza ca si algoritm QuickSort, cel ce utilizeaza
        //partitia Lomuto (implementata la inceputul lucrarii)
        void quicksort(int l, int u,int poz)
        {
            int i,m;
            if(l>=u) return;
            m=l;
            for(i=l+1;i<=u;i++)
            {
                if(vector[poz].categorie[l]>vector[poz].categorie[i])
                {
                    m++;

swap(vector[poz].categorie[i],vector[poz].categorie[m]);
                }
            }
            swap(vector[poz].categorie[l],vector[poz].categorie[m]);
            quicksort(l,m-1,poz);
            quicksort(m+1,u,poz);
        }

        void sep_hibrid()
        {
            char s[10];
            int i;
            for(i=1;i<=n;i++) //n reprezinta lungimea vectorului ce
                              //trebuie sortat
            {
                itoa(v[i],s,10); //transformam fiecare element din
                                 //vectorul v ce trebuie sortat in sir
                                 //de caractere pentru a-i afla
                                 //numarul de cifre (primul criteriu)

vector[strlen(s)].categorie[++vector[strlen(s)].lungime]=v[i];
            }
            for(i=1;i<=9;i++)
            {
                //pentru fiecare i reprezentand numarul de cifre
                //sortam toate numerele din categorie
                quicksort(1,vector[i].lungime, i);
            }
        }
```

Procedura *sep_hibrid* este usor de retinut si de inteles. Daca aveti un QuickSort gata implementat se poate spune ca tot algoritmul are aceeasi proprietate.

In continuare se va prezenta algoritmul de afisare.

```
void afisare()
{
    int i,j;
    for(i=1;i<=9;i++)
          for(j=1;j<=vector[i].lungime;j++)
                printf("%d",vector[i].categorie[j]);
}
```

2.4.4. Analiza complexitatii si a consumului de memorie

Vom incepe cu calcularea complexitatii.

Consideram pentru fiecare categorie *existenta*(vector[i].lungime>0) un numar m_i reprezentand numarul de numere din acea categorie. Vom nota cu k numarul total de categorii.

Se spune ca $\sum_{i=1}^{k} m_i = n$.

Deoarece nu stim exact datele de intrare, si toate acestea au acelasi grad de probabilitate, atunci vom lua in considerare un caz mediu. Vom lua pentru fiecare m_i un m mediu, unde $m = n/k$. Complexitatea acestui algoritm va fi:

$O(k * m \log m + n + v)$, unde $v = 9$.

Rezolvam $k * m \log m + n + v = (k * n/k)\log(n/k) + n + v = n\log(n/k) + n + v \Leftrightarrow$ $n\log n - n\log k + n + v$, pentru ca algoritmul sa fie mai rapid decat Quicksort trebuie ca $n + v - n\log k < 0 \Rightarrow 1 - \log k < -v/n \Rightarrow \log k > (n+v)/n$. Pe aceasta formula se bazeaza viteza algoritmului. Relatia este valabila si exista pentru $k > 2$, astfel **spunem ca pentru** $k \in [3,9]$ **(interval de numere naturale) algoritmul este mai rapid decat Quicksort.**

Cazul defavorabil este atunci cand $k = 1$, complexitatea fiind O(n log n + n + v).

Pentru calcularea consumului de memorie consideram aceleasi notatii ca mai sus, astfel acesta va fi: $O(k \log m + v*m + 2*v + 1) \Leftrightarrow O(k \log(n/k) + v*n/k + 2*v + 1)$. Se observa ca **memoria folosita de SEP hibrid este mult mai mica decat cea folosita de catre SEP-ul "nativ".**

Algoritmul de afisare al acestui algoritm este mult mai rapid decat algoritmul de afisare al SEP-ului "nativ", complexitatea fiind O(n+9). Pentru n>>9 se face aproximatia n+9≈n. Se afirma in final ca acest algoritm are complexitatea O(n).

2.4.5. Avantaje si dezvantaje

In continuare se vor puncta avatanjele si dezavantajele

Avantaje: timp de executie mic, memoria folosita este mai mica decat cea a SEP-ului "nativ", usor de inteles, se poate adapta astfel incat sa se poata sorta lexicografic.

Dezavantaje: memoria folosita este mai mare decat cea a Quicksort-ului.

Din punct de vedere al timpului de executie spunem ca:

SEP < SEP hibrid < Quicksort < Bubblesort

Din punct de vedere al memoriei folosite putem spune ca:

Bubblesort < Quicksort < SEP hibrid < SEP

Rezumat

Am analizat pe parcursul acestei lucrari anumite metode de sortare, si am aplicat noi metode, evidentiind avantajele si dezavantajele lor. In anumite contexte, unde viteza este primordiala, va recomand solutiile noi propuse, deoarece, dupa cum ati observat din analiza, atat SEP, cat si SEP hibrid sunt foarte rapide (SEP hibrid avand un plus cand vorbim de consum de memorie, iar SEP avand un plus cand este vorba de timp de executie).

Bibliografie

"*Introducere in algoritmi*" de Thomas H.Cormen, Charles E. Leiserson, Ronald R. Rivest

"*Three Beautiful Quicksorts*", prezentare tinuta de Jon Bentley in cadrul Google