Colegiul Național "Mihai Eminescu"
Botoșani

Studiu asupra algoritmilor de sortare

Avădănei Andrei
Clasa a IX-a

Cuprins

1. Introducere

1.1 Descrierea lucrării

Aceasta lucrare este dedicata studiul asupra timpilor si memoriei folosite de diferite ale algoritmilor frecvent utilizati,cat si a imbunatatirii acestor valori prin propunerea de algoritmi noi.Vom prezenta aici si implementarile ce le ofera STL-ul(Standard Template Library) pentru a usura implementarile algoritmilor de sortare.

Fiecare algoritm prezentat va fi caracterizat prin :

- ➔ *Cazul Mediu* - situatia in care probabilitatea duratei executiei algoritmului se apropie de media timpului de executie al algoritmului respectiv
- ➔ *Cazul defavorabil* - situatia in care timpul de executie al algoritmului este cel mai ridicat
- ➔ *Memorie folosita* – memoria necesara sortarii vectorului
- ➔ *Stabilitate* – stabileste daca algoritmul prezentat nu are nevoie de o implementare speciala pentru vectorii ce au valori egale
- ➔ *Sortare descrecatoare* – liniile de cod ce trebuie modificate in implementarea algoritmului pentru a sorta vectorul descrescator
- ➔ *Sortare crescatoare* – liniile de cod ce trebuie modificate in implementarea algoritmului pentru a sorta vectorul descrescator

1.2 Necesitatea sortării

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).

Există diferiți algoritmi de sortare : *Bubble Sort , Selective Sort , Quick Sort , Heap Sort* etc. fiecare cu avantajele si dezavantajele sale. Fiecare algoritm prezentat se bazeaza pe o metoda de sortare :

- *Bubble Sort [interschimbare]*
- *Selection Sort [selectie]*
- *Insertion Sort [inserare]*
- *Shell Sort [inserare]*
- *Merge Sort [interclasare]*
- *Heap Sort [selectie]*
- *Quick Sort [partitionare]*

1.3 Elemente fundamentale

***Definiție** : „Prin sortare inelegem algoritmul prin care putem rearanja k elemente intr-o anumita ordine (de exemplu: in ordine lexicografica, ordine crescatoare).”*

Dupa cum bine stim, fiecare algoritm de sortare are nevoie de un anumit spatiu si un anumit timp pentru executarea algorimului. Pentru masurarea complexitatii vom folosi notatia complexitatii „***O(n)***” (n reprezinta numarul de comparatii ale algoritmului). Vom citi „**complexitate de n**”.De asemenea, vom folosi o notatie asemanatoare pentru a masura consumul memoriei.

2. Prezentarea algoritmilor de sortare

2.1 Bubble Sort

Cazul mediu : **O(N^2)**
Cazul defavorabil : **O(N^2)**
Memorie folosita : **O(1)**
Stabil : **DA**
Sortare descrescatoare : **a[i] < a[i+1]**
Sortare crescatoare : **a [i] > a[i+1]**

Descriere :

Sortarea prin metoda bulelor se considera drept una din cele mai putin efective metode de sortare dar cu un algoritm mai putin complicat.
Ideea de baza a sortarii prin metoda bulelor este in a parcurge tabloul de la stanga spre dreapta, fiind comparate elementele alaturate a[i] si a[i+1]. Daca vor fi gasite 2 elemente neordonate valorile lor vor fi interschimbate.

Parcurgerea tabloului de la stinga spre dreapta se va repeta atat timp cat nu vor fi intalnite elemente neordonate.

Implementare :

```
void bubble(int a[],int n)
{
    int i,schimbat,aux;
    do
    {
       schimbat = 0;
       for(i = 0; i < n-1; i++)  //parcurgem vectorul
         if(a[i] < a[i+1])       //daca valoarea i din vectorul a este
                                 //mai mica decat cea de pe pozitia i+1
         {                       //interschimbare
            aux = a[i];
            a[i] = a[i+1];
            a[i+1] = aux;
            schimbat = 1;
         }
    }while(schimbat);
}
```

2.2 Selection Sort

Cazul mediu : **O(N^2)**
Cazul defavorabil : **O(N^2)**
Memorie folosita : **O(1)**
Stabil : **DA**
Sortare descrescatoare : **min < a[j]**
Sortare crescatoare : **min > a[j]**

Descriere :

Acest algoritm selecteaza la fiecare pas`ul i cel mai mic element din vectorul de la pasul i+1 pana la n.Valoarea minima de la pasul i este pusa in vector la pozitia i,facandu`se intereschimbarea cu pozitia actuala a minimului.Nu este un algoritm indicat pentru vectorii mari, in majoritatea cazurilor oferind rezultate mai slabe decat *insertion sort si bubble sort* .

Implementare :

```
void selective(int a[],int n)
{
     int i,aux,min,minat;
     for(i = 0; i < n - 1;i++)
   {
      minat = i;
      min = a[i];

      for(j = i + 1;j < n;j++) //selectam minimul
                               //din vectorul ramas( de la i+1 la n)
      {
         if(min > a[j])        //sortare crescatoare
         {
           minat = j;          //pozitia elementului minim
           min = a[j];
         }
      }
      aux = a[i] ;
      a[i] = a[minat];         //interschimbare
      a[minat] = aux;
   }
}
```

2.3 Insertion Sort

Cazul mediu : **O(N^2)**
Cazul defavorabil : **O(N^2)**
Memorie folosita : **O(1)**
Stabil : **DA**
Sortare descrescatoare : **j > 0 && a[j - 1] < a[j]**
Sortare crescatoare : **j > 0 && a[j - 1] > a[j]**

Descriere :

Spre deosebire de alti algoritmi de sortare, sortarea prin insertie este folosita destul de des pentru sortarea tablourilor cu numar mic de elemente. De exemplu, poate fi folosit pentru a imbunatati rutina de sortare rapida.

Sortarea prin insertie seamana oarecum cu sortarea prin selectie. Tabloul este impartit imaginar in doua parti - o parte sortata si o parte nesortata. La inceput, partea sortata contine primul element al tabloului si partea nesortata contine restul tabloului. La fiecare pas, algoritmul ia primul element din partea nesortata si il insereaza in locul potrivit al partii sortate.

Cand partea nesortata nu mai are nici un element, algoritmul se opreste.

Implementare :

```
void insertion(int a[], int n)
{
     int i, j, aux;
     for (i = 1; i < n; i++)
     {
         j = i;
        while (j > 0 && a[j - 1] > a[j])
        {
            aux = a[j]; a[j] = a[j - 1]; a[j--] = aux;
        }
     }
}
```

2.4 Shell Sort

Cazul mediu : -
Cazul defavorabil : **O(N * log^2 N)**
Memorie folosita : **O(1)**
Stabil : **NU**
Sortare descrescatoare : **j >= h && a[j-h] < v**
Sortare crescatoare : **j >= h && a[j-h] > v**

Descriere :

Algoritmul *shell sort* este o generalizare a algoritmului *insertion sort*. La algoritmul insertion sort, pentru a insera un nou element în lista de elemente deja sortate, se deplasează fiecare element cu câte o poziţie spre dreapta atât timp cât avem elemente mai mari decât el. Practic fiecare element înaintează spre poziţia sa finală cu câte o poziţie.

Algoritmul *shell sort* lucrează similar, doar că deplasează elementele spre poziţia finală cu mai mult de o poziţie. Se lucrează în iteraţii. În prima iteraţie se aplică un insertion sort cu salt s1 mai mare decât 1. Asta înseamnă că fiecare element din şirul iniţial este deplasat spre stânga cu câte s1 poziţii atât timp cât întâlneşte elemente mai mari decât el.

Se repetă asemenea iteraţii cu salturi din ce în ce mai mici s2, s3, s4, etc. Ultima iteraţie se face cu saltul 1. Această ultimă iteraţie este practic un insetion sort clasic.

Principiul este că după fiecare iteraţie şirul devine din ce în ce “mai sortat”. Iar cum algoritmul insertion sort funcţionează cu atât mai repede cu cât şirul este mai sortat, per ansamblu vom obţine o îmbunătăţire de viteză.

Implementare :

```
void shell(long a[],long n)
{
    long i, j, k, h, v;
    long cols[] = {1391376, 463792, 198768, 86961, 33936, 13776, 4592,
                   1968, 861, 336, 112, 48, 21, 7, 3, 1};

    for (k = 0; k < 16; k++)            //parcurgem fiecare limita
    {
        h = cols[k];
        for (i = h; i < n; i++)         //insertion sort
        {
            v = a[i];
            j = i;
            while (j >= h && a[j-h] > v) //crescator
            {
                a[j] = a[j-h];
                j = j - h;
            }
            a[j] = v;
        }
    }
}
```

2.5 Merge Sort

Cazul mediu : **O(N log N)**
Cazul defavorabil : **O(N log N)**
Memorie folosita : **O(N)**
Stabil : **DA**
Sortare descrescatoare : **b[i] >= a[j]**
Sortare crescatoare : **b[i] <= a[j]**

Descriere :

In cazul sortarii prin interclasare vectorii care se interclaseaza sunt doua secvente ordonate din acelasi vector.

Sortarea prin interclasare utilizeaza metoda **Divide et Impera**:

- se imparte vectorul in secvente din ce in ce mai mici., astfel incat fiecare secventa sa fie ordonata la un moment dat si interclasata cu o alta secventa din vector corespunzatoare.
- practic interclasarea va incepe cand se ajunge la o secventa formata din doua elemente.

Aceasta odata ordonata se va interclasa cu o alta corespunzatoare. Cele doua secvente vor alcatui in subsir ordonat din vector mai mare care la randul lui se va interclasa cu subsirul corespunzator s.a.m.d.

Implementare :

```
void mergesort(int a[],int st, int m, int dr)
{
    int b[100];
    int i, j, k;

    i = 0; j = st;
    // copiem prima jumatate a vectorului a in b
    while (j <= m)
        b[i++] = a[j++];

    i = 0; k = st;
    // copiem inapoi cel mai mare element la fiecare pas
    while (k < j && j <= dr)
        if (b[i] <= a[j])
            a[k++] = b[i++];
        else
            a[k++] = a[j++];

    // copiem elementele ramase daca mai exista
    while (k < j)
        a[k++] = b[i++];
}

void merge(int a[],int st, int dr)
{
  if (st < dr)
  {
     int m = (st+dr)/2;
     merge(a,st, m);
     merge(a,m+1, dr);
     mergesort(a,st, m, dr);
  }
}
```

2.6 Heap Sort

Cazul mediu : **O(N log N)**
Cazul defavorabil : **O(N log N)**
Memorie folosita : **O(1)**
Stabil : **NU**
Sortare descrescatoare : **if (a[w+1]<a[w]) w++;**
if (a[v]<=a[w]) return;
Sortare crescatoare : **if (a[w+1]>a[w]) w++;**
if (a[v]>=a[w]) return;

Descriere :

Algoritmul *HeapSort* este cel mai slab algoritm de clasa **O(N log2N)**
Este mai slab (dar nu cu mult) decat algoritmii din familia QuickSort, dar are marele avantaj fata de acestia ca nu este recursiv.

Algoritmii recursivi ruleaza rapid, dar consuma o mare cantitate de memorie, ceea ce nu le permite sa sorteze tablouri de dimensiuni oricat de mari

HeapSort este un algoritm care "impaca" viteza cu consumul relativ mic de memorie.

Implementare:

```
void swap(int a[],int i,int j)
{
   int aux = a[i];
   a[i] = a[j];
   a[j] = aux;

}
void downheap(int a[],int v,int n)
{
    int w = 2 * v + 1;                    // primul descendent al lui v
    while (w<n)
    {
       if (w+1<n)                         // mai exista unul?
          if (a[w+1]>a[w]) w++;           //crescator
                                          // w este decendentul lui v

        if (a[v]>=a[w]) return;           //crescator

        swap(a,v, w);                     // interschimbam v cu w
        v = w;                            // continuam
        w = 2 * v + 1;
    }
}


void heapsort(int a[],int n)
{
    for (int v = n/2-1; v >= 0; v--)       //creem heap`ul
      downheap (a,v,n);

    while (n>1)
    {
      n--;
      swap(a,0, n);
      downheap (a,0,n);
    }
}
```

2.7 Quick Sort

Cazul mediu : **O(N log N)**
Cazul cel mai nefavorabil : **O(N^2)**
Memorie folosita : **O(log N)**
Stabil : **DA**
Sortare descrescatoare :
```
while(vector[min] > mijl) min++;
while(vector[max] < mijl) max--;
```
Sortare crescatoare :
```
while(vector[min] < mijl) min++;
while(vector[max] > mijl) max--;
```

Descriere :

Quick Sort este unul dintre cei mai rapizi si mai utilizati algoritmi de sortare pana in acest moment,bazandu`se pe tehnica "**divide et impera**".Desi cazul cel mai nefavorabil este **O(N^2)** ,in practica,*QuickSort* ofera rezultate mai bune decat restul algoritmilor de sortare din clasa "**O(N log N)**".

Implementare :

```
void qSort(int vector[],int st,int dr)
{
   int temp,min,max,mijl;

   mijl = vector[st+(dr-st)/2];                //luam mijlocul intervalului
   min = st; max = dr;
   do
   {
        while(vector[min] < mijl) min++;
        while(vector[max] > mijl) max--;
        if(min <= max)                         //interschimbare
        {
            temp = vector[min];
            vector[min++] = vector[max];
            vector[max--] = temp;
        }
    }while(min <= max);  //la fiecare pas sortam "mai bine" intervalul st-dr

    //cand numai avem ce face schimbam intervalul
    if(st < max) qSort(vector,st,max);  //crescator
    if(dr > min) qSort(vector,min,dr);  //crescator
}
```

3. Implementări cu ajutorul STL

Pentru a usura munca programatorului si implicit a imparti timpul alocat unei probleme in alte directii librariile standard ofera ajutor pentru a implementa cativa dintre algoritmii de mai sus.

In libraria standard a limbajului C este implementata o varianta naiva de Quicksort,iar STL-ul ofera programatorilor C++ o implementare a algoritmului Introsort, cat si functiile make_heap si sort_heap, pentru a putea implementa usor Heapsort.

3.1 Heap Sort

Implementare :

```
#include <algorithm>
#include <vector>
#include <cstdio>

using namespace std;

vector<int> v;

int main()
{
    int i, x, n;

    freopen("heapsort.in","r",stdin);
    freopen("heapsort.out","w",stdout);

     scanf("%d",&n);

     for(i = 0; i < n; i++ )
     {
         scanf("%d",&x);
         v.push_back(x);
     }

     make_heap(v.begin(),v.end());
     sort_heap(v.begin(),v.end());


     for(i = 0; i < n; i++)
     {
        printf("%d ",v[i]);
     }
    return 0;
}
```

3.2 Quick Sort

Implementare :

```
#include <stdlib.h>
#include <stdio.h>

#define MAX_N 10000
int n, v[MAX_N];

int cmp(const void* x,const void *y);

int main(int argc, char *argv[])
{
   int i;

   freopen("qsort.in","r",stdin);
   freopen("qsort.out","w",stdout);

   scanf("%d",&n);

   for(i = 0; i < n; i++ )
        scanf("%d",v+i);

   qsort(v,n,sizeof(v[0]),cmp);

    for(i = 0; i < n; i++)
    {
       printf("%d ",v[i]);
    }

    return 0;
}

int cmp(const void*x, const void *y )
{
   return *(int*)x - *(int*)y;
}
```

3.3 Intro Sort

Cazul mediu **: O(N log N)**
Cazul defavorabil **: O(N log N)**
Memorie folosita : **O(log N)**
Stabil : **DA**

Descriere :

Introsort sau introspective sort este una dintre cele mai bune metode de sortare a unui vector.Ea se bazeaza pe doi algoritmi studiati precedent(*quick sort si heap sort*).Algoritmul incepe sa lucreze cu QuickSort ,dar cand recursivitatea atinge o anumita adancime(bazata pe numarul elementelor ce urmeaza sa fie sortate) algoritmul continua sortarea cu ajutorul metodei HeapSort.Optimizarea adusa QuickSort`ului in aceasta situatie va fi alegerea pivotului pe baza "median of 3" valorilor : st,dr,mijl.

Implementare :

```
#include <algorithm>
#include <vector>
#include <cstdio>

using namespace std;

vector<int> v;

int main()
{
    int i, x, n;

    freopen("introsort.in","r",stdin);
    freopen("introsort.out","w",stdout);

     scanf("%d",&n);

     for(i = 0; i < n; i++ )
     {
         scanf("%d",&x);
         v.push_back(x);
     }

     sort(v.begin(),v.end());

      for(i = 0; i < n; i++)
      {
         printf("%d ",v[i]);
      }

    return 0;
}
```

4. LSD Radix Sort(Least significant digit radix sort)

4.1 Descriere

Cazul mediu : **O(N * k)**
Cazul defavorabil : **O(N * k)**
Memorie folosita : **O(N)**
Stabil : **DA**

k = lungimea medie a fiecarei chei(numar,cuvant)

Descriere :

LSD Radix Sort este una dintre cele mai rapide metode de sortare.Aceasta se bazeaza pe sortarea in functie de cea mai nesemnificativa cifra(least significant digit).

4.2 Algoritmul

Deoarece *Radix Sort* este un algoritm hibrid, ce nu are la baza o tehnica de comparare fiecare element se numeste „cheie”.Valorile sunt gandite ca siruri de caractere si procesate pe biti. Algoritmul sorteaza cheile dupa urmatoarea regula : cheile de lungime mai mica sunt asezate in fata celor de lungime mai mare,iar cele de aceeasi lungime sunt asezate in ordine lexicografica.

LSD Radix Sort poate sorta un vector cu valori intregi sau de tip string in ordine lexicografica.Deoarece fiecare numar/cuvant este considerat un sir de caractere, procesarea incepe de la cea mai nesemnificativa dintre valorile sirului de caractere.Astfel valorile la pasul k sunt comparate,sortate iar cele ce au valori egale sunt lasate in pace pentru a fi evaluate ulterior la un pas k+i .

4.3 Implementare

```
#define COUNT_N 256
#define BYTE 8
#define N_MAX 1000000

int a[N_MAX],b[N_MAX],count[COUNT_N],ind[COUNT_N];

void rad(int *a,int *b,int byte,int n)
{
  memset(count,0,sizeof(count));
  int i,Lm = COUNT_N - 1;
  for(i = 0; i < n; ++i)
      ++count[(a[i]>>byte)&Lm];

  for(i = 0;i < COUNT_N; ++i)
      ind[i] = ind[i-1] + count[i-1];
  for(i = 0; i < n; ++i)
      b[ind[(a[i]>>byte)&Lm]++] = a[i];
}

void radix(int *a,int n)
{
  rad(a,b,0     ,n);
  rad(b,a,  BYTE,n);
  rad(a,b,2*BYTE,n);
  rad(b,a,3*BYTE,n);
}
```

4.4 Avantaje și dezavantaje

Acest algoritm nu este perfect suferind la capitolul memorie alocata, ce este aproape dubla fata de *QuickSort* ,dar poate reprezenta asul din maneca cand nu se tine cont de aceasta ci doar de viteza algoritmului.

LSD Radix Sort poate fi o alternativa pentru algoritmii de sortare clasici , in special in momentele in care dependenta de o viteza cat mai ridicata de sortare este in primul plan iar timpul pentru o implementare nu este presant.

Avantaje :

- *Viteza vizibil imbunatatita*
- *Posibilitatea de a sorta diferite tipuri de valori(cuvinte,numere etc.)*

Dezavantaje :

- *Implementarea dificila*
- *Consumul de memorie alocat ridicat*

5. Rezumat

Pe parcursul acestei lucrari am analizat si comparat diferite metode de sortare asupra vectorilor precum si am adus idei alternative pentru aceste situatii.

Dupa cum s-a aratat in aceasta lucrare cazurile in care se doreste cautarea de solutii noi pentru viteze ridicate ale sortarii exista iar *LSD Radix Sort* este una dintre acestea putand micsora timpul foarte puternic.

Am oferit cateva implementari pentru situatiile in care timpul ce il aveti la dispozitie pentru o problema sau un proiect doreste o atentie in alte parti ale structurii sale,STL oferind oferind cateva functii ce usureaza implementarea *QuickSort-ului,HeapSort-ului si a IntroSort-ului*.

6. Bibliografie

„*Introducere in algoritmi*” de Thomas H.Corme,Charles E. Leiserson, Ronald R. Rivest”

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

„*Secretele problemelor de informatica*” , Milu Carmaciu

„*Programarea algoritmilor.Aplicatii in Pascal si C/C++*” , Manuela Coconea si Cristina Luca