Metode de sortare

Autor: Rinciog Octavian

Sortarea reprezinta una dintre cei mai utilizate metode de programare. Are utilizari de la domenii precum matematica(statistica matematica), pana la limbi(realizarea unor dictionare).De aceea se impune sa gasim cei mai convenabili algoritmi si sa prezentam avantajele si dezavantajele acestora. Ne vom rezuma la cei mai importanti (in cartea lui Donald Knuth –" Algoritmi de sortare" sunt prezentati peste 30 asemenea metode de programare).

Metodele de sortare se clasifica in *metode directe* si *metode avansate.* Metodele directe se bazeaza,, pe algoritmi de dificultate redusa, usor de gasit si de inteles. Metodele directe pe care le vom lua in considerare sunt **sortarea prin selectie (SelectSort), sortarea prin insertie (InsertSort) si sortarea cu bule (BubbleSort).**

Metodele avansate se bazeaza pe algoritmi putin mai complicati, dar care nu necesita unostinte avansate de algoritmica. Cateva din cele mai cunoscute sunt **sortarea rapida (QuickSort), sortarea prin interclasare (MergeSort) si sortarea cu micsorarea incrementului(Shell Sort).**

La fiecare metoda voi descrie algoritmul, il voi exemplifica in Pseudocod si C++(la unele metode si parametrizat), voi analiza complexitatea algoritmilor si voi propune un set de probleme.

Analizând **complexitatea algoritmilor** factorul cel mai important este timpul de executie exprimat în "O-mare". Trebuie sã folosim mai multe criterii pentru evaluarea timpului de executie al algoritmului de sortare cum ar fi, *numãrul de pasi(de atribuiri)* ai algoritmului si *numãrul de comparatii* dintre elemente necesare pentru a sorta *N* elemente

Al doilea factor important este **cantitatea (volumul) de memorie suplimentarã** folositã de un algoritm de sortare. Metodele se împart în trei tipuri: cei care sortezã pe loc, nu folosesc memorie suplimentarã, cu exceptie poate o micã tabelã sau stivã; cei care au o reprezentare de listã înlãntuitã deci folosesc *N* cuvinte de memorie în plus pentru pointerii listei; cei care au nevoie de memorie suplimentarã pentru o copie a tabelei initiale.

Recomand ca lucrarea sa fie parcursa pas cu pas, intrucat nu exista un algoritm perfect. In unele cazuri metodele directe sunt mai economicoase(mai putine elemente de comparat), in altele cele avansate sunt mai bune. De asemenea recomand incepatorilor ca sa utilizeze fiecare metoda.

1. Metode directe

1.1 BubbleSort Mergi sus

Acesta metoda se rezuma la a compara fiecare element cu celelalte, facandu-se interschimbarea daca elementul mai mare are indexul mai mic. Este cea mai simpla metode de sortare si nu necesita cunoasterea detaliata a limbajului de programare. Poate fi folosita cu succes de catre incepatori.

Algoritmul este urmatorul:

```
simple-sort (a[Nmax+ 1])
        pentru i← 1 pana la  n
                pentru  j ← i+1 pana la n
                        daca a[i]>a[j]
                                interschimba(a[i], a[j])
```

Algoritmul in C++ (neparamatrizat) este :

```
void BubbleSort(Tip a[Nmax+1],int Size)
{
  for(int i=0;i<Size;i++)
      for(int j=i+1;j<=Size;j++)
        if(a[i]>a[j])
        {
         Tip aux=v[i];
         a[i]=a[j];
         a[j]=aux;
        }
}
```

Bubble sort este o metodã de sortare simplã, eficientã pentru un numãr mic de elemente (mai putin de 15), dar nu pentru tabele mari. Nu necesitã foate multã memorie, dar este de douã ori mai lentã decât InsertionSort în aproape orice situatie. Timpul de executie depinde de input, adicã de ordinea initialã al elementelor. Dacã tabela este deja sortatã e nevoie de un singur pas, adicã *N-1* comparãri. În cazul cel mai nefavorabil sunt necesare $N \times (N-1)/2$ comparãri si $N \times (N-1)/2$ interschimbãri. Performanta algoritmului în caz general este mai greu de analizat dar este asemãnãtor cazului nefavorabil. Complexitatea timpului al Bubble sortului este $O(N^2)$.

Algoritmul in C++ (parametrizat) este :

```
Void Bubblesort_para(Tip a[Nmax+1],int size)
 {
   int ok=0,i;
    do{
        for( i=1;i<size;i++)
         if (a[i]>a[i+1]) {interschimba(a[i],a[i+1]);ok=1;}
    }while (ok==1);
}
```

In acest caz in cazul cel mai defavorabil numarul de comparatii este tot NC=n(n-1)/2 dar spre deosebire de celalalt algoritm este mai rapid in cazurile medii. (daca nu mai gaseste doua elemente de interschimbat algoritmul se opreste.)

BubbleSort nu foloseste alte structuri de date si din aceasta cauza este recomandabil in cazurile cu putine elemente de sortat si in cazul in care memoria este limitata.

Exemplu de sortare (sirul initial este {6, 5, 4, 3, 2, 1}):

Nr interschinbari	**6**	**5**	**4**	**3**	**2**	**1**
1	**5**	**6**	4	3	2	1
2	5	**4**	**6**	3	2	1
3	5	4	**3**	**6**	2	1
4	5	4	3	**2**	**6**	1
5	5	4	3	2	**1**	**6**
6	**4**	**5**	3	2	1	6
7	4	**3**	**5**	2	1	6
8	4	3	**2**	**5**	1	6
9	4	3	2	**1**	**5**	6
10	**3**	**4**	2	1	5	6
11	3	**2**	**4**	1	5	6
12	3	2	**1**	**4**	5	6
13	**2**	**3**	1	4	5	6
14	2	**1**	**3**	4	5	6
15	**1**	**2**	2	3	4	5

Probleme propuse :

1. Care este numãrul minim de interschimbãri care vor sorta permutarea 10 11 1 5 6 7 3 4 2 9 8
2. Arãtati continutul tabelei 5 89 10 11 23 78 4 2 3 -1 dupã a cincea interschimbare realizata prin metoda bulelor.
3. Cate operatii de comparatii se fac la primul exemplu?
4. Dupa ce studiati si celelalte metode de sortare directe aratati de ce BubbleSort este mai lenta decat InsertionSort si decat SelectionSort precum si numarul de comparatii si interschimbari facute inutil !

1.2 SelectionSort Mergi sus

Selectia directã este una dintre cele mai simple metode de sortare si va lucra foarte bine pentru tabele mici, fiecare element (înregistrare), este mutat cel mult o datã.. Implementarea algoritmului este simplu.

Algoritmul presupune ca la fiecare pas "i " sa se gaseasca elementul minim dintre a[i+1], a[i+2]…a[n] si se interschimba cu a[i].

Algoritmul este urmatorul:

```
Selection(a[nmax+1])
      Pentru i←1 pana la n-1
            {min←a[i]
Pentru  j←i+1 pana la n
                  Daca (a[j] < min) min←a[j]}

            Daca (a[i]>min) interschimba(a[i],a[j])
```

Algoritmul in C++ este :

```
Selection (type a[], int size)
   {
     int i, j;
     type min, t;
     For ( i=1; i<=size-1; i++ )
      {
           min = a[i];
        For ( j = i+1; j<= size; j++ )
        If ( a[j]<min ) min = a[j];
        If (a[i]>min) t =a[min]; a[min]=a[i]; a[i]=t;
      }
  }
```

Se cheltuie cel mai mult timp cu cãutarea elementului minim din partea nesortatã a tabelei.Cãutarea se face de la stânga la dreapta. Pe parcursul fiecãrui pas este necesar o interschimbare, deci numãrul interschimbãrilor pentru *N* elemente este: *N-1*. Numãrul total al comparatilor este:

$$\sum_{1}^{n-1}(n-i)=\frac{n(n-1)}{2}$$

deci este proportional cu N^2. Timpul de executie al algoritmului este de ordinul O(N 2)în caz mediu si nefavorabil, chiar si când tabela este deja în ordine sortatã (crescãtoare). Este o metodã stabilã.

Exemplu (sirul initial este : {5, 6, 1, 2, 4, 3 })

Nr.intersch	5	6	1	2	4	3
1	**1**	6	**5**	2	4	3
2	1	**2**	5	**6**	4	3
3	1	2	**3**	**6**	4	5
4	1	2	3	**4**	**6**	5
5	1	2	3	4	**5**	**6**

Probleme propuse:

1. Aratati ce contine vectorul {7 8 9 2 4 5 1 6} dupa 5 interschimbari.
2. Implementati algoritmul folosind structura repetitiva"while"

1.3 InsertionSort Mergi sus

Insertia directã apartine familiei de tehnici de sortare care se bazeazã pe metoda "jucãtorului de bridge". Este un algoritm aproape la fel de simplu ca Selection sort, dar poate mai flexibil. Considerãm elementele A[1]...A[i-1] fiind sortate, inserãm elementul A[i] în locul ce îi revine.

Fiind dat o tabelã *A* cu *N* elemente nesortate, parcurgem tabela si le inserãm fiecare element în locul propriu între celelalte elemente considerate sortate. Pentru fiecare *i = 2..N* , elementele *A[1]...A[i]* sunt sortate prin inserarea lui *A[i]* între lista elementelor sortate: *A[1]...A[i-1]*. Elementele aflate în stânga indexului sunt în ordine sortatã dar nu sunt în pozitia lor finalã. Tabela este sortatã complect când indexul ajunge la capãtul drept al tabelei.

Algoritmul este :

```
Insertion(a[nmax+1])
 Pentru i←2 pana la n
   J←i
  pana cand (a[j-1]>a[i])
   a[j]←a[j-1] j← j-1
      a[j]←a[i]
```

Algoritmul in C++ este :

```
Insertion ( type A[ ], int size )
      {
        int i, j;
        type v;
        For ( i = 2; i <= size; i++ )
          {
             v = A[i]; j = i;
             While ( A[j-1] > v )// mut elementele cu o pozitie in
fata
               { A[j] = A[j-1]; j --; }
             A[j] = v;// pun elem in poz lui
          }
   }
```

Insertion sort este un algoritm de sortare simplu, liniar: *O(N)* pentru tabele care contin *N* elemente aproape sortate. Timpul de executie al algoritmului depinde de numãrul inversãrilor, deci de ordinea initialã al elementelor. În caz general sunt necesare $N^2/4$ comparatii si interschimbãri, $N^2/8$ deci ordinea magnitudinii este $O(N^2)$. În caz cel mai nefavorabil, când tabela este initial sortatã în ordine inversã mutarea elementelor se realizeaza de

$$\sum_{i=2}^{n}(i-1)=\frac{n*(n-1)}{2}=\frac{(n*n-n)}{2}$$

ori, sortarea este de $O(N^2)$ .Dacã tabela *A* contine înregistrãri mari atunci e preferabil folosirea unei *"tabele de index"*: *P*, accesând tabela originalã doar pentru comparare, astfel costul este mai putin. Altã modalitate de îmbunãtãtire constã în folosirea unei matrici de pointeri pentru înregistrãri.

Exemplu (sirul initial este :{5,6,7,10,1,2})

Nr intersch	5	6	7	10	1	2
1	1	5	6	7	10	2
2	1	2	5	6	7	10

Probleme propuse :

1. Arãtati continutul tabelei: 5 6 8 9 10 15 4 7 3 -1 dupã a patra interschimbare a realizata prin metoda insertiei directã.
2. Sã se gãseascã timpul de rulare minim si maxim în functie de N, pentru programul care foloseste algoritmul InsertionSort .

2. Metode avansate de sortare

2.1 QuickSort Mergi sus

În practicã algoritmul de sortare cel mai rapid este Quicksort numitã **sortare rapidã**, care foloseste partitionarea ca idee de bazã. Este mai rapidã decât orice metodã de sortare simplã, se executã bine pentru fisiere sau tabele mari, dar ineficient pentru cele mici.Strategia de bazã folositã este *"divide et impera"*, pentru cã este mai usor de sortat douã tabele mici, decât una mare. Algoritmul este usor de implementat, lucreazã destul de bine în diferite situatii si consumã mai putine resurse decât orice altã metodã de sortare în multe situatii. Necesitã numai în jur de *NlogN* operati în cazul general pentru a sorta *N* elemente.

Metoda QuickSort presupune gasirea pozitiei finale pe care o ocupa elemenetul de pe prima pozitie comparandu-l cu elementele din cealalta partitie a tabelului, acest algoritm realizandu-se pana cand partitia are 1 element.

Astfel pentru urmatoarea configuratie a tabelului initial: 6 5 4 3 7 2 9 10
QuickSort va sorta vectorul astfel:

6	5	4	3	7	2	9	10	
i							j	6<10
6	5	4	3	7	2	9	10	
i						j		6<9
6	5	4	3	7	2	9	10	
i					j			schimba(6,2)
2	5	4	3	7	6	9	10	
	i				j			5<6
2	5	4	3	7	6	9	10	
		i			j			4<6
2	5	4	3	7	6	9	10	
			i		j			3<6
2	5	4	3	7	6	9	10	
				i	j			schimba(7,6)
2	5	4	3	6	7	9	10	
				j	i			stop(j<i)

Procedeul se va repeta cu cele 2 partitii ((1,j-1) si (i,n) pana cand fiecare partitie va avea un element:

2	3	4	5	6	7	9	10

Algoritmul este :

```
Imparte (st,dr,p)
{
i←st
j←dr
ii←0
jj←-1
  cat timp (i<j)
       { daca (a[i]>a[j]) {aux←a[i]    a[i]←a[j]   a[j]←aux
                           aux←ii     ii← -jj       jj←-aux}
          i←i+ii
          j←j+jj}
}

QuickSort(st,dr)
   {
   Daca st<dr { imparte (st,dr,p) // se det pozitia p pe care o
ocupa primul element
         QuickSort (st,p-1);
         QuickSort (p+1,dr);}

   }
```

Algoritmul in C++ este :

```
Quicksort(type A[], int st, int dr)
    {
      int i,j;
      type v,t;
      If ( dr>st )
        {
          v=A[dr]; i=st-1; j=dr;     /*indice=ultimul element */
          For ( ; ; )
           {
             While ( A[++i]<v );
             While ( A[--j]>v );
             If (i>=j) break;
             t=A[i]; A[i]=A[j]; A[j]=t;
           }
          t=A[i]; A[i]=A[r]; A[r]=t;
          Quicksort(A,st,i-1);
          Quicksort(A,i+1,dr);
        }
   }
```

Potrivit algoritmului, fiecare element este comparat cu pivoul, adicã operatiunea este de *O(N)*, tabela este divizatã în douã pãrti, fiecare parte este divizatã iarãsi în douã…. Dacã fiecare parte este împãrtitã aproximativ în jumãtate, va rezulta $log_2 N$ împãrtiri. Deci timpul de executie al Quicksortului în caz mediu este de $O(N \log_2 N)$, iar în caz nefavorabil $O(N^2)$.

Quicksort este o metodã bunã în caz general, dar nu si în caz nefavorabil când este preferabil folosirea a 3 indicii de impartire. Randomizarea este o idee importantã si folositoare, o unealtã generalã pentru a îmbunãtãti algoritmul. Quicksort este sensibil la ordinea datelor de intrare. Nu este o metodã stabilã.

Dezavantajul algoritmului este cã, e recursiv. Necesitã în jur de N^2 de operatii în caz nefavorabil. Este fragil, o simplã gresealã în implementare poate cauza o executare gresitã pentru anumite fisiere.

Probleme propuse :

1. Scrieti un program care combinã algoritmii de sortare Quicksort cu Bubblesort astfel: utilizeazã Quicksort pentru obtinerea de partitii nesortate de lungime m, 1< m < n, apoi utilizeazã Bubblesort pentru terminarea sortãrii.
2. Scrieti un program care combinã algoritmii de sortare Quicksort si SelectionSort astfel: utilizeazã Quicksort obtinerea de partitii nesortate de lungime m, 1< m < n, apoi utilizeazã selectare directa pentru terminarea sortãrii.

2.2 MergeSort Mergi sus

Algoritmul de sortare prin interclasare se bazeazã pe urmãtoarea idee: pentru a sorta o tabelã cu N elemente îl împãrtim în douã tabele pe care le sortez separat si le intrclasãm. Este o metodã de sortare care foloseste strategia de bazã "divide et impera", conform cãreia problema se descompune în alte douã subprobleme de acelasi tip si dupã rezolvarea lor rezultatele se combinã. Algoritmul sorteazã elementele în ordine crescãtoare.

Tabelul se imparte in n/2, dupa aceea tabelele se impart in jumatate, tot asa pana cand tabelele formate au mai putin sau cel mult de k elemente(in cazul nostru k=2-este cel mai usor sa compari 2 elemente).

Algoritmul presupune :

1. Afla mijlocul tabelului
2. Sorteaza prima jumatate a tabelului
3. Sorteaza a doua jumatate a tabelului
4. Imbina cele 2 jumatati

Algoritmul in Pseudocod este:

```
Mergesort(st,dr)
    {Daca (dr>st)
       M←(st+dr)/2
       Mergesort(st,m)
       Mergesort(m+1,dr)
         Pentru i←m+1 pana la 1   b[i-1]=a[i-1]
         Pentru j←m pana la dr    b[m+dr-st]=b[j+1]
         Pentru k=1 pana la dr
                  {daca (b[i]<b[j]) a[k] =b[i]
                    altfel a[k]=b[j]
                    i←i+1
                    j←j-1}
    }
```

Algoritmul in C++ este :

```
Mergesort(int A[], int l, int r)
    {
       int i,j,k,m;
       If (r>l)
         {
           m=(r+1)/2;
           Mergesort(A,l,m);
           Mergesort(A,m+1,r);
           For (i=m+1; i>l; i--)   B[i-1]=A[i-1];
           For (j=m; j<r; j++)   B[m+r-j]=A[j+1];
           For (k=l; k<=r; k++)
              A[k] = (B[i]<B[j]) ? B[i++] : B[j--];
         }
  }
```

Majoritatea muncii este cuprinsã în comparare si copiere. În procerura Merge fiecare element din subtabele este comparat: asta este o operatie de *O(N)*, copierea elementelor din tabela temporalã în cea originalã tot *O(N)*. Deci procedura Merge este de *O(N)*. Procedura Mergesort este recursivã si Merge este apelat recursiv pentru ambele jumãtãti. Astfel putem împãrti tabela de $log_2 N$ ori si de fiecare datã executãm de *O(N)* procedura Merge. Algoritmul total este de $O(N \log_2 N)$ în toate cele trei cazuri. Este o metodã stabilã. Nu este sensibil la ordinea initialã al elementelor. Dezavantajul Mergesortului este cã necesitã o tabelã auxiliarã de mãrimea originalului. Problema apare când tabela este mare si spatiu este un factor critic.

Probleme propuse :

1. Câte comparatii va folosi Mergesort pentru a sorta cheile: 4,6,5,1,4,7,-1,-2,8,9 ?
2. Modificati algoritmul Mergesort ca sã obtineti un algoritm de sortare prin interclasarea listelor cu douã cãi.
3. Scrieti un program care foloseste algoritmul de sortare prin interclasare cu douã cãi pentru N=1000 de înregistrãri, câmpul cheie fiind de tip sir de caracter .
4. Comparati algoritmele bazate pe „divide et impera" – MergeSort si QuickSort prin implementarea pe un sir format din 3500 elemente.(acestea se pot da prin folosirea functiei „random()")

2.3 ShellSort Mergi sus

Sortarea cu micsorarea incrementului (shellsort) este o extensie simplã al Insertion sortului care câstigã vitezã permitând schimbarea elementelor aflate departe. Ideea de bazã o constituie rearanjarea elementelor din tabela A în asa fel încât, luând fiecare a h-a element (începând de oriunde), sã obtinem o tabelã sortatã. Astfel spunem cã tabela este h-sortatã. O tabelã , h-sortatã este formatã din h subtabele sortate independent, intercalate. Folosind astfel o procedurã pentru fiecare secventã a valorii lui h care se terminã în 1, va produce o tabelã sortatã.

Algoritmul este :

```
shell-sort (a)
      h ← 1
      pana cand h<n/9
            h ← 3*h+1
      pana cand h>0
            pentru i ← h+1 pana la n
                  t ← a[i]
                  j ← i
                  pana cand (j>h si a[j-h] > t)
                        interschimba(a[j], a[j-h])
                        j ← j-h
            h ← h/3
```

Algoritmul in C++ este :

```
Shellsort(int A[], int n)
   {
     int i, j, h, v;
     for (h = 1; h <= n/9; h = 3*h+1);
     for ( ; h > 0; h /= 3 )
     for (i = h+1; i <= n; i += 1)
      {
        v = A[i]; j = i;
        while ( j>h && A[j-h]>v )
          {
            A[j] = A[j-h];
            j - = h;
          }
        A[j] = v;
      }
   }
```

Exemplu : sirul initial este {1 11 8 10 12 4 7 3 1 13 1 6 9 5 2}

Pasuri	1 11 8 10 12 4 7 3 1 13 1 6 9 5 2
Pas 1	**1** **<u>11</u>** 8 10 12 4 7 3 1 13 1 6 9 **5** **<u>2</u>** 1 **2** 8 10 12 4 7 3 1 13 1 6 9 5 **11**
Pas 2	**<u>1</u>** <u>2</u> <u>8</u> 10 **<u>12</u>** <u>4</u> <u>7</u> 3 **<u>2</u>** <u>13</u> <u>1</u> 6 **<u>9</u>** <u>5</u> <u>11</u> **1** 2 8 10 **2** 4 7 3 **9** 13 1 6 **12** 5 11 1 **2** 8 10 2 **4** 7 3 9 **5** 1 10 12 **13** 11 1 2 **1** 10 2 4 **7** 3 9 5 **8** 10 12 13 **11** 1 2 1 **3** 2 4 7 **6** 9 5 8 **10** 12 13 11 1 2 1 3 2 4 7 6 9 5 8 10 12 13 11
…	.....
Pas M	**1 1 2 2 3 4 5 6 7 8 9 10 11 12 13**

Acest program foloseste secventa de crestere (incrementare):1093, 364, 121, 40, 13, 4, 1 care este destul de eficient. Există si alte secvente de crestere: de exemplu….64, 32, 16, 8, 4, 2, 1 care au performante slabe, pentru cã elementele de pe pozitiile pare nu sunt comparate cu cele de pe pozitiile impare pânã la sfârsit.

Cele *N* elemenete se împart la prima trecere în [N/2] grupe, care la a doua trecere se reunesc douã câte douã, formându-se [N/4] grupe s.a.m.d. La fiecare trecere grupele se sortezã separat.(Notatia […] semnificã parte întreagã)

Eficienta algoritmului Shell sort este greu de analizat, greu de comparat cu alte metode, nici timpul de rulare nu este cunoscut sigur. Pentru acest program în legãturã cu timpul de rulare existã douã presupuneri : $N(\log_2 N)^2$ si $N^{3/2}$. Timpul de executie nu este sensibil în mod particular la ordinea initialã al elementelor. Insertion sort este mai lent

pentru cã schimbã doar elementele adiacente. Shellsort nu efectuazã mai mult de $N^{1/2}$ comparatii (pentru cresterile: 1, 4, 13, 40, 121,….). Shellsort este metoda aleasã pentru multe aplicatii de sortare pentru cã are un timp de executie accesibil chiar si pentru o tabelã moderat de mare (mai putin de 5000 elemente) si este o metodã stabilã.

Probleme propuse :

1. Sã se gãseascã un algoritm eficient pentru construiera permutãrilor care vor maximiza numãrul de deplasãri în algoritmul Shellsort.
2. Dati un exemplu pentru a arãta de ce 8, 4, 2, 1 nu ar fi o alegere bunã de a termina o secventã de crestere Shellsort.

3.Concluzii : Mergi sus

Algoritmul	**Compexitatea**
Insertion Sort	$O(n^2)$
Buble Sort	$O(n^2)$
SelectionSort	$O(n^2)$
Shell Sort	$O(n^{3/2})$ (estimat)
Merge Sort	$O(n\log_2 n)$
Quick Sort	$O(n\log_2 n)$ - caz mediu $O(n^2)$ - caz defavorabil

Metodele directe sunt mai bune pentru tabelurile cu putine elemente (<100), pntru ca cele avansate sunt ineficiente pe aceste tabele, dar pentru vectorii cu multe elemente metodele avansate sunt foarte eficiente. Una dintre cele mai bune metode este ShellSort, dar timpul de executie este foarte greu de analizat, asadar pentru tabele mari este mai bun QuickSort.

4. Bibliografie Mergi sus

1. http://jukebox.tg-mures.roedu.net/Volumes/cd_educational/CD/SORT/1fremes_index.htm

2. http://sorting.trusign.ro/SA/default.htm

3. http://www.lgcip.home.ro/index/C_C__/Probleme/body_probleme.html

4. http://www.ginfo.ro

5. Informatica-Analiza pentru grupele de performanta – Clasa X – Ed.Dacia 2004

4.1 Pentru a vedea appleturile dedicate sortarii accesati :

1. http://www.cs.bme.hu/~gsala/alg_anims/2/sortchoiceinp.html

2. http://www.cs.pitt.edu/~kirk/cs1501/animations/SortApplets.html