Algoritmi de sortare

Operaţia de ordonare a unor articole în funcţie de diverse criterii este foarte des întâlnită în practică. Se cunosc o mulţime de algoritmi de sortare, majoritatea fiind foarte simpli. În continuare vom încerca să prezintem unele dintre cele mai cunoscute metode de sortare.

Metodele de sortare cele mai des folosite pot fi clasificate în două categorii: metode directe şi metode avansate.

Metodele directe se bazează pe algoritmi de dificultate redusă, uşor de găsit şi de înţeles. Metodele directe pe care le vom lua în considerare sunt sortarea prin selecţie (SelectSort), sortarea prin inserţie (InsertSort) şi sortarea cu bule (BubbleSort).

Metodele avansate se bazează pe algoritmi puţin mai complicaţi, dar care nu necesită cunoştinţe avansate de algoritmică. Câteva din cele mai cunoscute sunt sortarea rapidă (QuickSort), sortarea prin interclasare (MergeSort) şi sortarea cu ansamble (HeapSort).

Orice programator trebuie să cunoască metodele de sortare şi să aleagă folosirea unei metode, în funcţie de criteriul de eficienţă urmărit. Eficienţa metodelor de sortare, după cum am precizat mai sus pentru algoritmi în general, se măsoară după tipul de execuţie şi memoria folosită.

<u>Sortarea prin selecţie sau Select Sort</u>

Algoritmul constă în alegerea celui mai mic element dintr-un vector şi aşezarea lui pe prima poziţie, repetată pentru şiruri din ce în ce mai scurte. Metoda necesită un timp de lucru care depinde de numărul de elemente din vector, iar algoritmul metodei se reprezintă prin structuri repetitive cu număr cunoscut de paşi.

În cazul unui vector sortat crescător, primul element (cu indice 1) este cel mai mic dintre cele cu indici de la **1** la **n**, cel de-al doilea este cel mai mic dintre cele cu indici de la **2** la **n** ş.a.m.d.

Să considerăm un vector în care elementele cu indici de la **1** la **i-1** sunt deja sortate. Pentru a continua procesul de sortare, dintre elementele rămase (cu indici de la **i** până la **n**) trebuie găsit cel mai mic (cu indice **isel**) şi adus în poziţia **i**.

	i-1	i		isel		n
. . .			. . . .		. . . .	
elemente sortate		Elemente nesortate				

Exemplu:

I	Vectorul prelucrat							isel
1	10	5	6	12	3	7	12	9	5
2	3	5	6	12	10	7	12	9	2
3	3	5	6	12	10	7	12	9	3
4	3	5	6	12	10	7	12	9	6
4	3	5	6	7	10	12	12	9	8
6	3	5	6	7	9	12	12	10	8
7	3	5	6	7	9	10	12	12	7

În acest exemplu valoarea minimă dintr-o secvenţă este marcată prin haşurare, iar locul în care trebuie plasată aceasta este marcat prin chenar dublu.

*Algoritmul **SelSort** (X, N)*

```
pentru 1 <= i < n
   { determina  isel = indicele valorii minime (i <= isel <= n)
     daca isel > i atunci
       inverseaza x[i] cu x[isel], folosind zona tampon temp;
   }
```

Sortarea prin insertie

În cazul sortării prin inserţie se consideră că vectorul este sortat pană la o anumită poziţie şi se încearcă inserarea următorului element pe poziţia potrivită. Dacă vectorul ar avea un singur element, el ar fi deja sortat; în cazul unui vector cu 2 elemente, care nu respectă relaţia de ordine, este suficientă inversarea acestora. Să presupunem că vectorul are mai mult de 2 elemente şi că am sortat deja, în ordine crescătoare, primele **i-1** elemente. Pentru a extinde sortarea la **i** elemente este necesară deplasarea la dreapta, cu o poziţie, a tuturor elementelor mai mari decât cel cu indicele **i**, urmată de inserarea sa în poziţia corespunzătoare. Aceasta presupune compararea elementului inserat cu cele deja sortate, situate la stânga sa. Dacă această operaţie se efectuează de la dreapta spre stânga, atunci ea poate fi combinată cu cea de deplasare.

	J	j+1				i-1	i	
. . .				. . . . . .				. . .
<= temp		> temp → deplasate dreapta					temp	

Exemplu:

i	Vectorul prelucrat							
1	10	5	6	12	3	7	12	9
2	5	10	6	12	3	7	12	9
3	5	6	10	12	3	7	12	9
4	5	6	10	12	3	7	12	9
5	3	5	6	10	12	7	12	9
6	3	5	6	7	10	12	12	9
7	3	5	6	7	10	12	12	9
8	**3**	**5**	**6**	**7**	**9**	**10**	**12**	**12**

În acest exemplu valoarea care trebuie inserata într-o secvenţă deja sortată este marcată cu un chenar dublu, iar valorile mai mari decât ea, care trebuie deplasate spre dreapta, sunt marcate prin haşurare.

Algoritmul ***InserSort*** *(x, n)*

```
pentru 1 <= i < n
  { copiaza x[i] in temp;
    deplaseaza la dreapta toate valorile > temp, cu indici j, i-1 >= j >= 0
    inlocuieste ultimul element deplasat cu temp
  }
```

O metodă de sortare mai eficientă şi foarte des folosită este **bubble sort** sau sortarea prin metoda bulelor. Această metodă presupune parcurgerea vectorului şi compararea elementelor alăturate (a[i] şi a[i+1]); dacă ele nu respectă ordinea dorită, îşi vor interschimba valorile. Vectorul va fi parcurs de atâtea ori, până când va fi sortat, adică nu se va mai efectua nici o interschimbare de valori. În acest scop va fi folosită o variabilă de tip întreg, care să „ţină minte” dacă s-au mai efectuat interschimbări. Secvenţa de algoritm corespunzătoare este:

```
main()
{int a[20], n, i, aux, gata=0;
 .................................
 while (!gata)
   {gata=1;
    for (i=0; i<n-1; i++)
       if (a[i]>a[i+1])
           {aux=a[i];
            a[i]=a[i+1];
            a[i+1]=aux;
            gata=0;
           }
     }
 .................................
 }
```

Dacă vectorul **a** conţinea la început aceleaşi elemente: 7, 2, 9, 1, 3, iată câţiva paşi din execuţia algoritmului:

gata=0,

atâta timp cât gata este zero:

gata =1

i=0 este a[0]>a[1]? da, deci se interschimbă cele două valori şi gata devine zero; **a** devine: 2, 7, 9, 1, 3

i=1 este a[1]>a[2]? nu, deci nu se întâmplă nimic;

i=2 este a[2]>a[3]? da, deci se interschimbă cele două valori şi gata rămâne zero; **a** devine: 2, 7, 1, 9, 3

i=3 este a[3]>a[4]? da, deci se interschimbă cele două valori şi gata rămâne zero; **a** devine: 2, 7, 1, 3, 9

Se reia structura while, deoarece gata este zero:

gata =1

i=0 este a[0]>a[1]? nu, deci nu se întâmplă nimic;

i=1 este a[1]>a[2]? da, deci se interschimbă cele două valori şi gata devine zero; **a** devine: 2, 1, 7, 3, 9

i=2 este a[2]>a[3]? da, deci se interschimbă cele două valori şi gata rămâne zero; **a** devine: 2, 1, 3, 7, 9,

i=3 este a[3]>a[4]? nu, deci nu se întâmplă nimic;

Se reia structura while, deoarece gata este zero:

gata =1

i=0 este a[0]>a[1]? da, deci se interschimbă cele două valori şi gata devine zero; **a** devine: 1, 2, 3, 7, 9

i=1 este a[1]>a[2]? nu, deci nu se întâmplă nimic;

i=2 este a[2]>a[3]? nu, deci nu se întâmplă nimic;

i=3 este a[3]>a[4]? nu, deci nu se întâmplă nimic;

Se reia structura while, deoarece gata este zero:
gata =1
Se mai parcurge o dată vectorul, dar nu se mai efectuează nici o intershimbare, căci acesta este ordonat crescător. Condiţia de menţinere în structura repetitivă nu mai este îndeplinită şi algoritmul se încheie. Metoda necesită un timp de lucru care depinde de numărul de treceri prin vector şi de numărul de interschimburi la fiecare trecere.

Sortarea prin metoda bulelor îmbunătăţită

Aceasta metodă se bazează pe faptul că într-un vector sortat crescător toate perechile de valori consecutive trebuie să respecte relaţia de ordine. Dacă această relaţie nu este respectată, atunci valorile trebuie interschimbate. Verificarea perechilor de valori trebuie reluată până când nu mai este necesară nici o interschimbare, dar fiecare nouă etapă de verificare se poate opri la perechea din faţa celei care a necesitat ultima interschimbare (**u_inv**), deoarece, evident, perechile următoare respectă relaţia de ordine.

Exemplu:

lim	ip	Vectorul prelucrat							u_inv
6	0	**10**	**5**	**6**	**12**	**3**	**7**	**12**	0
	1	5	10	6	12	3	7	12	1
	2	5	6	10	12	3	7	12	1
	3	5	6	10	12	3	7	12	3
	4	5	6	10	3	12	7	12	4
	5	5	6	10	3	7	12	12	4
4	0	5	6	10	3	7	12	12	0
	1	5	6	10	3	7	12	12	0
	2	5	6	10	3	7	12	12	2
	3	5	6	3	10	7	12	12	3
3	0	5	6	3	7	10	12	12	0
	1	5	6	3	7	10	12	12	1
	2	5	3	6	7	10	12	12	1
1	0	5	3	6	7	10	12	12	0
0		**3**	**5**	**6**	**7**	**10**	**12**	**12**	

Fiecare pereche analizată este evidenţiată printr-un chenar dublu, haşurat în cazul în care elementele trebuie interschimbate. Limita fiecărei etape de verificare este marcată prin linie dublă mai groasă.

*Algoritmul **B_Sort** (x, n)*

```
lim = n - 1;
cat timp lim > 0
{  u_inv = 0
   pentru fiecare pereche ip, 0 <= ip < lim
      daca ordine incorecta atunci { interschimba valorile din perechea ip;
                                     u_inv = ip;
                                   }
   lim = u_inv;
 }
```

Algoritmul de sortare prin interschimbare

Este unul din cei mai simpli algoritmi de sortare, dar nu şi cel mai eficient.

Consideraţii teoretice.

Se consideră dată o secvenţă S={s1,s2,..,sn} de n elemente, pe care este definită o relaţie de ordine liniară<. Iniţial ordinea elementelor în cadrul secvenţei S este aleatoare. Scopul sortării este aranjarea elementelor într-o nouă secvenţă Sn={sn1, sn2,..,snn}, astfel încât sni < sni+1 pentru i=1..n-1. Algoritmul are la bază următoarea metodă de sortare: fiecare element din vector sni, 0<=i<n-1, este comparat cu toate elementele situate în vector după el, snj, i<j<n; în cazul în care cele două elemente nu sunt în ordinea dorită, acestea se vor interschimba.

```
main()
{int a [20], n, i, j, aux;
 ……………………..
 for (i=0; i<n/1; i++)
   for (j=i+1; j<i; j++)
      if (a[i]>a[j])
         {aux=a[i];
          a[i]=a[j];
          a[j]=aux;
         }
  ………………………..
  }
```

Dacă vectorul **a** conţinea la început elementele: 7, 2, 9, 1, 3, iată câţiva paşi din execuţia algoritmului:

i=0, j=1 este a[0]>a[1]? da, deci se vor interschimba primele 2 elemente:

a devine: 2, 7, 9, 1, 3

i=0, j=2 este a[0]>a[2]? nu, deci nu se întâmplă nimic;

i=0, j=3 este a[0]>a[3]? da, deci se vor interschimba elementele a[0] şi a[3]:

a devine: 1, 7, 9, 2, 3

i=0, j=4 este a[0]>a[4]? nu, deci nu se întâmplă nimic;

Se trece la următoarea valoare pentru **i**, deoarece **for**-ul pentru **j** s-a terminat:

i=1, j=2 este a[1]>a[2]? nu, deci nu se întâmplă nimic;

i=1, j=3 este a[1]>a[3]? da, deci se vor interschimba elementele a[1] şi a[3]:

a devine: 1, 2, 9, 7, 3

În final, componentele vectorului vor fi: 1, 2, 3, 7, 9

Metoda de programare *divide et impera*

Constă în împărţirea problemei iniţiale de dimensiuni [n] în două sau mai multe probleme de dimensiuni reduse. În general se execută împărţirea în două subprobleme de dimensiuni aproximativ egale şi anume [n/2] . Împărţirea în subprobleme are loc până când dimensiunea acestora devine suficient de mică pentru a fi rezolvate în mod direct (cazul de bază). După rezolvarea celor două subprobleme se execută faza de combinare a rezultatelor în vederea rezolvării întregii probleme .

Metoda *divide et impera* se poate aplica în rezolvarea unei probleme care îndeplineşte următoarele condiţii:

- se poate descompune în (două sau mai multe) subprobleme;
- aceste subprobleme sunt independente una faţă de alta (o subproblemă nu se rezolvă pe baza alteia şi nu se foloseşte rezultatul celeilalte);
- aceste subprobleme sunt similare cu problema iniţială;
- la rândul lor subproblemele se pot descompune (daca este necesar) în alte subprobleme mai simple;
- aceste subprobleme simple se pot soluţiona imediat prin algoritmul simplificat.

Deoarece puţine probleme îndeplinesc condiţiile de mai sus, aplicarea metodei este destul de rară. După cum sugerează şi numele "desparte si stăpâneşte" paşii algoritmului sunt:

Pas1. Descompunerea problemei iniţiale în subprobleme independente (care folosesc mulţimi de date de intrare disjuncte - d_i), similare problemei iniţiale, de dimensiuni mai mici;

Pas2. Dacă subproblema permite rezolvarea imediată (corespunde cazului de bază), atuncise rezolvă obţinându-se soluţia s; altfel se revine la *Pas1*.

Pas3. Se combină soluţiile subproblemelor în care a fost descompusă (s_i) o subproblemă, până când se obţine soluţia problemei iniţiale.

Deoarece subproblemele în care se descompune problema sunt similare cu problema iniţială, algoritmul *divide et impera* poate fi implementat recursiv. Subprogramul recursiv **divide-et-impera(d,s)**, unde **d** reprezintă dimensiunea subproblemei (corespunde mulţimii datelor de intrare), iar **s** soluţia subproblemei, poate fi descris în pseudocod astfel:

```
divide-et-impera(d,s)
început     dacă dimensiunea d corespunde unui caz de bază
                atunci de determină soluţia s a problemei;
                astfel
                     pentru i=1,n execută
                               se determină dimensiunea d_i a subproblemei P_i;
                               se determină soluţia s_i a subproblemei P_i prin
                               apelul divide-et-impera(d_i,s_i) ;
                      sfârşit_pentru;
                     se combină soluţiile s_1, s_2, s_3, …, s_n;
                     sfârşit_dacă;
sfârşit;
```

Implementarea acestui algoritm în limbajul C++ se face astfel:

/*declaraţii globale pentru datele de intrare ce vor di divizate în submulţimi disjuncte pentru subproblemele în care se descompune problema*/

```
void divizeaza (<parametri: submulţimile>)
{// se divizează mulţimea de date de intrare în submulţimi disjuncte d_i}
void combina (<parametri: soluţiile s_i care se combină>)
{// se combină soluţiile obţinute s_i}
void dei (<parametri: mulţimea de date d şi soluţia s>)
{// declaraţii de variabile locale
 if (<este caz de bază>) {// se obţine soluţia corespunzătoare subproblemei}
    else
        {divizeaza (<parametri: k submulţimi>);
          for (i=1; i=k; i++)
            dei (<parametri: mulţimea de date d_i şi soluţia s_i>);
          combină (<parametri: soluţiile s_i>);}}
void main ()
{// declaraţii de variabile locale
 // se citesc datele de intrare ale problemei – mulţimea d
    dei (<parametri: mulţimea de date d şi soluţia s>);
// se afişează soluţia problemei - s}.
```

Metoda *divide et impera* se recomandă în următoarele cazuri:

- algoritmul obţinurt este mai eficient decât algoritmul clasic (iterativ) – de exemplu, algoritmul de căutare într-un vector sortat şi algoritmii pentru sortarea unui vector;
- rezolvarea problemei prin diivzarea ei în subprobleme este mai simplă decât rezolvarea clasică(iterativă) .

Sortarea rapidă (QuikSort)

Prin această metodă de sortare se execută următoarele operaţii prin care sunt rearanjate elementele din cadrul vectorului:

x_1	x_2	x_3		x_{i-1}	x_i	x_{i+1}		X_{n-1}	x_n

Pivotul Vectorul iniţial

- Primul element din vector, numit *pivot,* este mutat în cadrul vectorului pe poziţia pe care trebuie să se găsească în vectorul sortat.

➢ Toare elementele mai mici decât el vor fi mutate în vector în faţa sa.

➢ Toate elementele mai amri decât el vor fi mutate în vector după el.

x_s	X_{s+1}	X_{s+2}		x_{m-1}	X_m	x_{m+1}		X_{d-1}	X_d

Subvectorul din stânga pivotului Pivotul Subvectorul din stânga pivotului

(elemente mai mici decât pivotul) (elemente mai mari decât pivotul)

De exemplu, dacă vectorul conţine elementele {3, 4, 1, 5, 2}, după executarea operaţiilor precizate vectorului va fi {2, 1, 3, 5, 4}.

Folosind metoda *divide et impera* problema iniţială va fi descompusă în subprobleme, astfel:

Pas1. Se rearanjează vectorul, determinându-se poziţia în care va fi mutat pivotul (m).

Pas2. Problema iniţială (sortarea vectorului iniţial) se descompune în două subprobleme prin descompunerea vectorului în doi subvectori: vectorului din stânga pivotului şi vectorul din dreapta pivotului, care vor fi sortaţi prin aceeaşi metodă. Aceşti subvectori, la rândul lor, vor fi şi ei rearanjaţi şi împărţiţi de pivot în doi subvectori etc.

Pas3. Procesul de descompunere în subprobleme ca continua până când, prin descompunerea vectorului în subvectori, se vor obţine vectori care conţin un singur element.

Subprogramele specifice algoritmului *divide et impera* vor avea următoarea semnificaţie:

➢ În subprogramul **divizeaza()** se va rearanja vectorul şi se va determina poziţia pivotului x_m, care va fi folosită pentru divizarea vectorului în doi subvectori :[x_s, x_{m-1}] şi [x_{m+1}, x_d].

➢ Subprogramul **combina()** nu mai este necesar, deoarece combinarea soluţiilor se face prin rearanjarea elementelor în vector.

În subprogramul **divizeaza()** vectorul se parcurge de la ambele capete către poziţia în care trebuie mutat pivotul. Se vor folosi doi indici: **i** – pentru parcurgerea vectorului de la începutul lui către poziţia pivotului (i se va incrementa) şi **j** – pentru parcurgerea vectorului de la sfârşitul lui către poziţia pivotului (j se va decrementa). Cei doi indici vor fi iniţializaţi cu capetele vectorului (i=s, respectiv j=d) şi se vor deplasa până când se întâlnesc, adică atât timp cât i<j. În momentul în care cei doi indici s-au întâlnit înseamnă că operaţiile de rearanjare a vectorului s-au terminat şi pivotul a fost adus în poziţia corespunzătoare lui în vectorul sortat. Această poziţie este i (sau j) şi va fi poziţia m de divizare a vectorului.

Voi prezenta în continuare exemple unde se vor utiliza două versiuni pentru subprogramul divizează().

Versiunea 1. Se folosesc variabilele logice: **pi**, pentru parcurgerea cu indicele i, şi **pj**, pentru parcurgerea cu indicele j. Ele au valoarea: 1 – se parcurge vectorul cu acel indice, şi 0 – nu se parcurge vectorul cu acel indice; cele două valori sunt complementare.

```
#include<iostream.h>
int x[100], n;
void schimb(int &a, int &b) {int aux=a; b=aux;}
void divizeaza(int s, int d, int &m)
{int I=s, j=d, pi=0, pj=1;
               //pivotul fiind pe pozitia s, parcurgerea incepe cu indicele j
while(I<j)
      {if (x[i], x[j]); schimb(pi,pj);}
       i=i+pi; j=j-pj;}
m=i;}
void QuikSort(int s, int d)
{int m;
if (s<d) {divizeaza(s,d,m);
            QuikSort (s,m/1);
            QuikSort (m+1,d);}}
void QuikSort()
{int I; cout<<"n= ";cin >>n;
      for (i=1;i<=n;i++) {cout<<"x["<<I<<"]= ";cin>>x[i];}
QuikSort (1,n);
cout<<"vectorul sortat"<< endl; for (i=1;i<=n;i++) cout<<x[i]<<" ";}
```

<u>Exemplu</u> .<u>*Versiunea 1*</u>

i				j
1	2	3	4	5
3	4	1	5	2

Cei doi indici i şi j sunt iniţializaţi cu extremităţile vectorului(i=1, j=5) şi parcurgerea începe cu indicele j (pi=0, pj=1)

	i			j
1	2	3	4	5
2	4	1	5	**3**

Se compară pivotul (3) cu ultimul element (2). Deoarece pivotul este mai mic, cele două variabile se interschimbă, şi se schimbă şi modul de parcurgere (pi=1; pj=0 – avansează indicele i).

	i		j	
1	2	3	4	5
2	**3**	1	5	4

Se compară elementul din poziţia i(4) cu elementul din poziţia j(3). Deoarece 4 este mai mare decât 3, cele două valori se interschimbă, şi se schimbă şi modul de parcurgere (pi=0;pj=1 – avansează inicele j).

	i	j		
1	2	3	4	5
2	**3**	1	5	4

Se compară elementul din poziţia i(3) cu elementul din poziţia j(5). Deoarece 3 este mai mic decât 5, cele două valori nu se interschimbă, şi se schimbă modul de parcurgere (pi=0;pj=1 – avansează inicele j).

		ij		
1	2	3	4	5
2	1	**3**	5	4

Se compară elementul din poziţia i(3) cu elementul din poziţia j(1). Deoarece 3 este mai mare decât 1, cele două valori se interschimbă, şi se schimbă modul de parcurgere (pi=0;pj=1 – avansează inicele i).Cei doi indici fiind egali, algoritmul se termină.

Versiunea 2

i				j
1	2	3	4	5
3	4	1	5	2

Iniţial, cei doi indici i şi j sunt iniţializaţi cu extremităţile vectorului(i=1, j=5) şi pivotul are valoarea 3.

i				j
1	2	3	4	5
2	4	1	5	**3**

Elementul din poziţia i(3) nu este mai mic decât pivotul; indicele i nu avansează (i=1). Elementul din poziţia j(2) nu este mai mare decât pivotul; indicele j nu avansează(j=5). Valorile din poziţiile i şi j se interschimbă.

	i			j
1	2	3	4	5
2	**3**	1	5	4

Elementul din poziţia i(2) este mai mic decât pivotul; indicele i avansează până la primul element mai mare decât pivotul (i=2). Elementul din poziţia j(3) nu este mai mare decât pivotul; indicele j nu avansează(j=5). Valorile din poziţiile i şi j se interschimbă.

	i	j		
1	2	3	4	5
2	1	**3**	5	4

Elementul din poziţia i(3) nu este mai mic decât pivotul; indicele i nu avansează (i=2). Elementul din poziţia j(4) este mai mare decât pivotul; indicele j avansează până la primul element mai mic decât pivotul (j=3). Valorile din poziţiile i şi j se interschimbă.

		j	i	
1	2	3	4	5
2	1	**3**	5	4

Elementul din poziţia i(1) este mai mic decât pivotul; indicele i avansează până la primul element mai mare decât pivotul (i=4). Elementul din poziţia j(3) nu este mai mare decât pivotul; indicele j nu avansează(j=3), algoritmul se termină.

MergeSort - sortare prin divizare şi interclasare (fuziune)

În cazul acestei metode vectorul de sortat este divizat în subvectori, prin înjumătăţiri succesive, cât timp lungimea acestora este > 2. Evident, un subvector de lungime 1 este sortat, iar un subvector de lungime 2 necesită cel mult interschimbarea celor doua valori. Monotoniile din subvectorii sortaţi sunt interclasate succesiv, în ordinea inversă divizării, obţinând în final vectorul sortat. Deoarece interclasarea necesită un vector auxiliar, sortarea propriu-zisă va fi precedată de alocarea unei zone tampon de aceeaşi lungime cu vectorul de sortat.

Pentru a evita copierea rezultatului unei interclasari din vectorul auxiliar în cel de sortat şi a reduce astfel numărul de operaţii, vectorul iniţial şi cel auxiliar pot fi utilizaţi alternativ ca sursă şi, respectiv, rezultat al operaţiei de interclasare. De asemenea, dacă analizăm exemplele următoare:

monotonii sursa		rezultat interclasare
| 15, 18 | 3, 8, 11 |	=>	| 3, 8, 11, 15, 18 |
| 2, 5 | 5, 14 |	=>	| 2, 5, 5, 14 |

observăm că ultimul element dintr-o monotonie este mai mic sau egal cu primul din cealaltă monotonie. În astfel de cazuri interclasarea poate fi înlocuită prin copierea celor două monotonii, în ordine inversă (primul exemplu) sau păstrând ordinea existentă (al doilea exemplu).

Exemplu:

Vectorul prelucrat								sursa, rez	lung. monotonii
10	**5**	**6**	**12**	**3**	**7**	**12**	**9**	**w**, v	8
10	5	6	12	3	7	12	9	**v**, w	4, 4
10	5	6	12	3	7	12	9	**w**, v	2, 2, 2, 2
5	10	6	12	3	7	9	12	w, **v**	2, 2, 2, 2
5	6	10	12	3	7	9	12	v, **w**	4, 4
3	**5**	**6**	**7**	**9**	**10**	**12**	**12**	w, **v**	8

În acest exemplu frontierele monotoniilor sunt marcate cu linie dubla, iar monotoniile care implică interschimbare sau interclasare sunt evidenţiate prin haşurare.

Algoritmul MergeSort (v, n)

```
creeaza w - copia vectorului v
MSort (n, w, v) - sorteaza cele n elemente din w, obtinand rezultatul in v
```

Algoritmul MSort (n, sursa, rez)

```
daca n este
  1: revenire;
  2: daca primul element din sursa > al doilea atunci
        copiaza in rez in ordine inversa (al doilea, primul)
  altfel
     determina dimensiunea primului subvector (j = n / 2)
     apeleaza MSort pentru sortarea primelor j elemente din rez in sursa;
     apeleaza MSort pentru sortarea ultimelor n-j elemente din rez in sursa;
     daca ultimul din prima monotonie din sursa <= primul din a doua monotonie atunci
        copiaza cele n elemente din sursa in rez
     altfel
        daca ultimul element din sursa <= primul element atunci
           copiaza in rez elementele din sursa aflate in a doua monotonie, si apoi
           cele din prima monotonie
        altfel
           interclaseaza cele doua monotonii din sursa in rez
```

Algoritmul HeapSort

HeapSort este unul din algoritmii de sortare foarte performanţi şi mai este cunoscut sub denumirea de “sortare prin metoda ansamblelor”. Deşi nerecursiv, este aproape la fel de performant ca si algoritmii de sortare recursivi (QuickSort fiind cel mai cunoscut). HeapSort este un algoritm de sortare “in situ”, adică nu necesită structuri de date suplimentare, ci sortarea se face folosind numai spaţiul de memorie al tabloului ce trebuie sortat. Exista şi implementari HeapSort care nu sunt “in situ”.

Algoritmul se aseamănă, în unele privinţe, cu sortarea prin selecţie (SelSort). La fiecare pas, cel mai mic element din tablou este găsit şi mutat în spatele tabloului, fiind ignorat de paşii următori, care vor continua pe restul tabloului. Diferenţa faţă de SelSort este că paşii următori ai algoritmului vor depune un efort mai mic (chiar mult mai mic) pentru a depista minimul din tabloul rămas. Fiecare pas al algoritmului are darul de a uşura sarcina paşilor ce urmează, ceea ce duce la performanţa foarte bună a algoritmului.