1.Sortarea prin inserare directă

Deşi algoritmul de sortare prin inserare directă este de ordin n^2, el este important pentru evidenţierea principiilor de bază privind sortarea şi indexarea *multicriterială.*

Principiul algoritmului constă în a considera al doilea element al şirului şi de a-l ordona în raport cu primul, apoi de a extrage al treilea element şi de a-l insera în şirul ordonat al primelor două şi aşa mai departe, până la inserarea ultimului element în şirul ordonat format cu primele *n-1* elemente.

12	1	7	2	5	8	11	9	18	14	6	25	21	15	16
1	12	**7**	2	5	8	11	9	18	14	6	25	21	15	16
1	7	12	2	5	8	11	9	18	14	6	25	21	15	16
1	2	7	12	5	8	11	9	18	14	6	25	21	15	16
1	2	5	7	12	8	11	9	18	14	6	25	21	15	16
1	2	5	7	8	12	11	9	18	14	6	25	21	15	16
1	2	5	7	8	11	12	9	18	14	6	25	21	15	16
1	2	5	7	8	9	11	12	18	14	6	25	21	15	16
1	2	5	7	8	9	11	12	18	14	6	25	21	15	16
1	2	5	7	8	9	11	12	14	18	6	25	21	15	16
1	2	5	6	7	8	9	11	12	14	18	25	21	15	16
1	2	5	6	7	8	9	11	12	14	18	25	21	15	16
1	2	5	6	7	8	9	11	12	14	18	25	21	15	16
1	2	5	6	7	8	9	11	12	14	15	18	25	21	16
1	2	5	6	7	8	9	11	12	14	15	16	18	25	21

Extragerea unei componente a şirului este echivalentă cu stocarea ei într-o variabilă temporară.

Inserarea acestei componente în subşirul sortat, implică deplasarea ascendentă a valorilor mai mari din subşirul sortat în vederea eliberării unei locaţii, în care este apoi memorată componenta extrasă.

Operaţiile de extragere componentă, deplasare ascendentă şi inserare în locaţia liberă apar ca faze distincte în funcţia *Sort*.

1	2	7	12		5
1	2	2	7	12	5
1	2	5	7	12	5

```
// Sortarea prin insertie directa
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
//typedef int (*PFCMP) (const void *, const void *);
typedef unsigned char BYTE;

     void sort(int x[], int n);

     void main()
     {
     clrscr();
      int k; int d=14;
      int tab[]={22,31,11,19,15,9,16,14,20,77,23,2,4,7};
      sort(tab, d);

          printf(" \n");
              for(k=0;k<d;k++)
              printf("%3d  ",tab[k]);
              getchar();
       }
      void sort(int x[], int n)
      {
      int xi;
      int i, j, k;

      for (i=1; i<n ; i++) {
          xi=x[i];
          j=i-1;
          while ((j>=0) && (x[j]>xi)) {
          x[j+1]=x[j];
          j--;
          }
          x[j+1]=xi;
           for(k=0;k<n;k++)
              printf("%3d",x[k]);
              printf("%6d \n",xi);
           }
       }
```

2.Sortarea după două criterii

Dacă se urmăreşte sortarea a două şiruri x[] şi y[] folosind ca prim criteriu mărimea elementelor primului şir şi drept criteriu secundar mărimea elementelor celui de-al doilea şir, în cazul în care x[i] = x[j] se apelează la compararea y[i] ? y[j], care decide ordinea elementelor în şirul principal x[1..n].

Procedeul poate fi generalizat pentru oricâte şiruri. În acest caz însă, există un procedeu mai eficient şi anume sortarea prin indexare.

```
// Sortarea unui sir dupa doua criterii
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>

     void sort(int x[], int y[], int n);

     void main()
     {
     clrscr();
      int k; int d=14;
      int tabx[]={22,31,11,19,15, 9,19,14,20,19,11, 2,11, 7};
      int taby[]={1 , 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14};
      sort(tabx,taby, d);

          printf(" \n");
              for(k=0;k<d;k++)
              printf("%3d  ",tabx[k]);  printf(" \n");
               for(k=0;k<d;k++)
              printf("%3d  ",taby[k]);  printf(" \n");
              getchar();
       }
      void sort(int x[], int y[], int n)
      {
      int xi, yi;
      int i, j, k;

      for (i=1; i<n ; i++) {
          xi=x[i];
          yi=y[i];
          j=i-1;
          while ((j>=0) && ((x[j]>xi)||((x[j]==xi)&&(y[j]>yi))))
{
          x[j+1]=x[j];
          y[j+1]=y[j];
          j--;
          }
          x[j+1]=xi;
          y[j+1]=yi;
           //   for(k=0;k<n;k++)
           //      printf("%3d",x[k]);
           //      printf("%6d \n",xi);
           }
       }
```

3.Sortarea prin indexare

Să considerăm un şir X_i, $i=1, 2, 3, \ldots n$, care trebuie utilizat într-un numit context în ordinea crescătoare a elementelor sale. În locul sortării efective, se poate construi, mai simplu, un vector de indici I_i, $i=1, 2, 3, \ldots n$, astfel încât şirul:

$$X_{I_i},\ i=1, 2, 3, \ldots n,$$

să fie ordonat crescător în raport cu indicele i . Vectorul I va conține indicii componentelor lui X ordonat crescător, adică I_1 va fi indicele celui mai mic element iar I_n , indicele celui mai mare element din X.

Exemplu:

Înainte de sortarea indexata:							
X=	21	17	12	27	11	14	28
I=	I_1=1	I_2=2	I_3=3	I_4=4	I_5=5	I_6=6	I_7=7

După sortarea indexată:							
X=	21	17	12	27	11	14	28
I=	I_1=5	I_2=3	I_3=6	I_4=2	I_5=1	I_6=4	I_7=7

Tabloul de indici I conține toată informația asupra ordonării crescătoare a şirului X şi poate fi folosit pentru referirea indirectă a elementelor, oricând este necesară utilizarea acestora în ordinea prescrisă de criteriu.

Indexarea este foarte utilă mai ales atunci când trebuie realizată exploatarea unui şir în raport cu mai multe criterii de ordonare. În acest caz, se construieşte câte un tablou de indici pentru fiecare criteriu de ordonare.

Eficiența sortării prin indexare este cu atât mai mare cu cât elementele şirului X ocupă un număr mai mare de locații de memorie, adică atunci când acestea sunt date de tip structură.

Algoritmul de construire a unui tablou indicial necesită ca prim pas, aşa după cum se poate vedea din funcția **`Index`** dată mai jos, inițializarea componentelor **`ind[i]`** cu valorile **`i`**, ca pentru cazul unui şir deja sortat. Se efectuează apoi o sortare prin inserare directă, însă orice operație care trebuie efectuată asupre componentelor taloului **`x`**, este efectuată de fapt aupra componentei corespunzătoare a tabloului indicial **`Ind`**.

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
     void Index(int x[], int ind[], int n);
     void main()
     {
     clrscr();
      int k; int d=14; int ind[20];
      int tab[]={0,22,31,11,19,15,9,16,14,20,77,23,2,4,7};
      Index(tab, ind, d);
          printf("\n Tablou initial de indici: \n");
              for(k=1;k<=d;k++)
              printf("%3d  ",k);
          printf("\n Tablou nesortat \n");
              for(k=1;k<=d;k++)
              printf("%3d  ",tab[k]);
          printf("\n\n\n Tablou de indici: \n");
              for(k=1;k<=d;k++)
              printf("%3d  ",ind[k]);
          printf("\n Tablou sortat \n");
              for(k=1;k<=d;k++)
              printf("%3d  ",tab[ind[k]]);

              getchar();
       }

      void Index(int x[], int ind[], int n)
   {
      int xi;
      int i, j, k;
      for (i=1; i<=n; i++) ind[i]=i;  // initializare
      for (i=2; i<=n; i++) {
          xi=x[i];                 // salveaza comp. de inserat
          j=i-1;
          while ((j>0) && (x[ind[j]]>xi)) {   //translateaza
          ind[j+1]=ind[j];
          j--;
          }
          ind[j+1]=i;                          //insereaza
           }
       }
```

Se observă că referirea la elementele din subşirul sortat **x[ind[j]]** se realizează prin intermediul tabloului de indici. Această adresare indirectă nu este necesară pentru componenta **x[i]**, care urmează a fi inserată în subşirul sortat, deoarece **ind[i]=i** la iniţializare iar la paşii anteriori au fost modificate doar componentele **ind[j], cu j<i**.

Odată construit tabloul de indici **ind** pentru un anumit şir **x**, el poate fi utilizat şi pentru indexarea altor tablouri în raport cu mărimea componentelor tabloului **x**.

În cazul în care este necesară indexarea simultană în raport cu două criterii ierarhizate – mărimea componentelor unui tablou **x** (criteriu principal) şi respectiv, mărimea componentelor unui tablou **y** (criteriu secundar) se poate utiliza funcţia **Index2**:

```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>
     void Index2(int x[], int y[], int ind[], int n);
     void main()
     {
     clrscr();
      int k; int d=15; int ind[20];
int tabx[]={0,22,31,11,19,15,9,19,14, 20, 19, 11,  2, 11, 7};
int taby[]={0,1,2,13, 4, 5, 6,  7,  8,  9, 10,  7, 12,  3,14};
      Index2(tabx,taby,ind, d);

          printf(" \n");
           for(k=1;k<d;k++)
              printf("%3d  ",tabx[k]);  printf(" \n");
           for(k=1;k<d;k++)
              printf("%3d  ",taby[k]);  printf(" \n\n\n");
           for(k=1;k<d;k++)
              printf("%3d  ",tabx[ind[k]]);  printf(" \n");
           for(k=1;k<d;k++)
              printf("%3d  ",ind[k]);   printf(" \n");
              getchar();
       }
      void Index2(int x[], int y[], int ind[], int n)
      {    int xi, yi;
           int i, indj, j;
      for (i=1; i<n ; i++) ind[i]=i;
      for (i=1; i<n ; i++)  {
          xi=x[i];  yi=y[i];
          j=i-1;    indj=ind[j];
          while ((j>0) &&
((x[indj]>xi)||((x[j]==xi)&&(y[indj]>yi)))) {
          ind[j+1]=ind[j];
          indj=ind[--j];
          }
          ind[j+1]=i;
           //   for(k=0;k<n;k++) printf("%3d",x[k]);
           //      printf("%6d \n",xi);
           }
       }
```

4.Tehnica “Divide et impera”

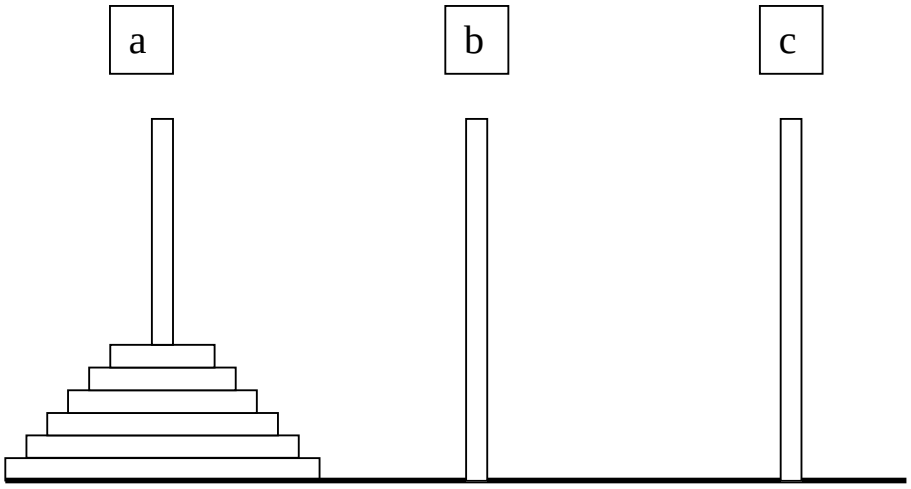
Se bazează pe principiul descompunerii unei probleme complexe în mai multe probleme mai simple (care la rândul lor se descompun) iar soluția generală se obține prin reunirea rezultatelor parțiale.

Un exemplu de utilizare a tehnicii “Divide et impera” îl constituie metoda Quick-sort de sortare internă, în care şirul de sortat este divizat în două şiruri, apoi în 4 etc.

Problema turnurilor din Hanoi

Se consideră 3 tije notate *a, b, c*. Pe tija *a* se găsesc *n* – discuri de diametre diferite aşezate în ordine crescătoare a diametrelor de sus în jos. Se cere să se mute discurile de pe tija *a* pe *b*, cu ajutorul tijei *c*, respectând următoarele reguli:

- la o mutare, se deplasează un singur disc;
- nu se permite aşezarea unui disc cu diametrul mai mare peste unul cu diametrul mai mic.



Pentru n=1 soluția este mutarea *a-b*, adică se mută discul de pe *a* pe *b*. Pentru n=2, se fac mutările: *ac, ab, cb*.

Pentru *n>2*, situația se complică; notăm cu H(*n, a, b, c*) şirul mutărilor celor *n* discuri de pe tija *a* pe tija *b* utilizând ca ajutor *c*.

Conform strategiei, descompunem problema în trei mai simple:

- mutarea a *(n-1)* discuri de pe *a* pe *c* (cu ajutorul lui *b*)
- mutarea discului rămas, pe *b* (mutarea *ab*);
- mutarea a (*n-1*) discuri de pe *c* pe *b* (cu ajutorul lui *a*);

Descompunerea de mai sus se poate exprima formal prin:
H(*n*, *a*, *b*, *c*)=H(*n-1*, *a*, *c*, *b*), ***ab***, H(*n-1*, *c*, *b*, *a*), pentru n> 1.

La rândul lor, cele două şiruri de mutări se descompun după aceeaşi regulă până se ajunge la *n=2*, caz în care soluția este simplă.

În programul de mai jos, algoritmul este implementat recursiv şi se generează şirul de mutări pentru un *n* dat.

```
#include<stdio.h>
#include<stdlib.h>
     void han(int , char , char , char );
     int con=1;

     void main()
     {
          char x, y, z;
          int nr;
          printf("Introduceti numarul de discuri: n= ");
          scanf("%d",&nr);
          x='a'; y='b'; z='c';
          han(nr, x, y, z);
          printf("\n\n");

     }

     void han(int n, char a, char b, char c)
     {
          if (n==1) {
               if (con % 25 ==0) printf("%c%c \n",a,b);
               else printf("%c%c ",a,b);
               con++;}
          else {
               han(n-1, a,c,b);
//             printf("%c%c ",a,b);
               if (con % 25 ==0) printf("%c%c \n",a,b);
               else printf("%c%c ",a,b);
               con++;
               han(n-1, c,b,a);}
          }
```