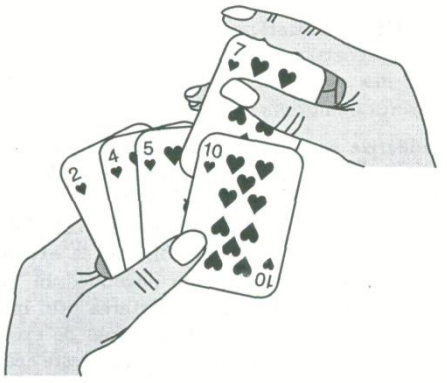
SORTAREA PRIN INSERARE

- STUDIU -

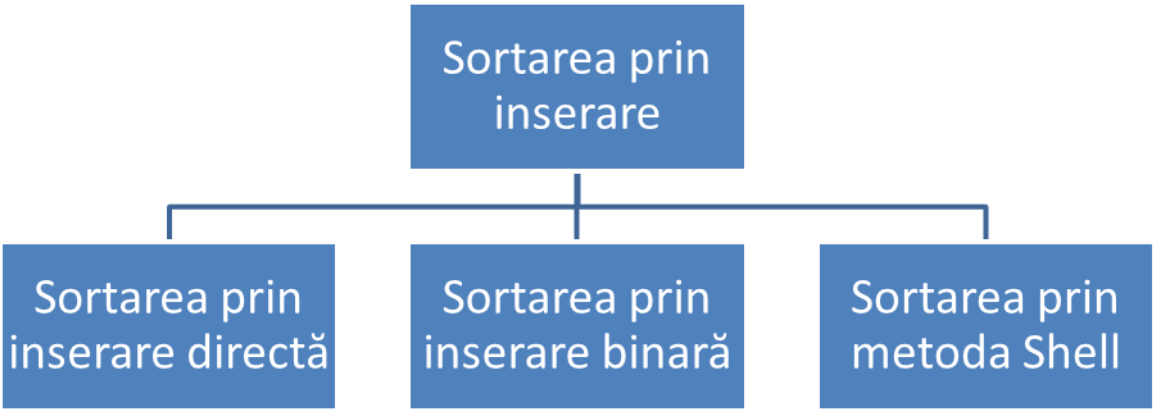
Autor: Secită Neli

Ideea algoritmului este de a considera pe rând fiecare element al tabloului şi de a-l insera în subtabloul ordonat până la momentul respectiv pe locul său. Acest loc s-ar putea să nu fie definitiv, deoarece este posibil ca următorul element să se insereze undeva în faţa acestuia.

Principiul sortării prin inserare îl putem asemăna cu sortarea unui pachet de cărţi de joc. Se aşează pachetul de cărţi pe masa cu faţa în jos şi apoi se ia câte o carte care este inserată în poziţia corectă. Pentru a găsi poziţia corectă pentru o carte dată, o comparăm cu fiecare dintre cărţile aflate deja, de la dreapta la stânga, aşa cum este ilustrat în următoarea figură:



Modul de sortare a unei mâini de cărţi, utilizând sortarea prin inserare



SORTAREA PRIN INSERARE DIRECTĂ

PREZENTAREA METODEI

Se consideră indicele j ($1 < j \leq n$) ce corespunde elementului care urmează să fie inserat. Elementele v_1, v_2, ..., v_{j-1} sunt elementele care sunt deja aranjate astfel încât $v_1 \leq v_2 \leq ... \leq v_{j-1}$. Indicele i se deplasează de la dreapta la stânga în interiorul vectorului. Plecând de la poziţia j-1, elementele sunt succesiv, deplasate o poziţie spre dreapta până când este găsită poziţia corectă pentru v_j, moment în care acesta este inserat. Elementele de intrare sunt sortate pe loc, în cadrul aceluiaşi vector v.

Deoarece v_j va fi aşezat la locul său, această metodă de sortare a fost denumită metoda de cernere (sifting) sau scufundare (sinking).

PREZENTAREA ALGORITMULUI

Algoritmul de sortarea prin inserare directă va sorta elementele şirului $v_1, v_2, \ldots, v_n$, iar după sortare, elementele vectorului vor fi în ordinea $v_1 \leq v_2 \leq ... \leq v_n$.

Pseudocodul pentru sortarea prin inserţie directă este reprezentat prin subprogramul numit ***inserare_directa*** care are doi parametri:

- **n** prin care primeşte un număr natural ce reprezintă lungimea vectorului
- **v** prin care primeşte un vector care memorează un şir de **n** numere care trebuie sortat

Algoritmul în pseudocod este următorul:

```
Subalgoritm inserare_directa (n,v)
  pentru j =2,n executa
      element = v[j]
      i = j -1
      cat timp((i>=1) si (v[i]>element)) executa
                  v[i+1] = v[i]
                  i = i-1
      sfârşit cat timp
      v[i+1] = element
  sfârşit pentru
sfârşit subalgoritm
```

Exemplu

Modul de operare a sortării prin inserţie directă asupra vectorului v este următorul:

V[1]	V[2]	V[3]	V[4]	V[5]	V[6]
3	8	10	7	9	12

Elementul ce urmează a fi inserat — **Ordinea elementelor în urma inserării**

V[2]
8

V[1]	V[2]
3	8

V[3]
10

V[1]	V[2]	V[3]
3	8	10

Elementul ce urmează a fi inserat — **Ordinea elementelor în urma inserării**

V[4]
7

V[1]	V[2]	V[3]	V[4]
3	7	8	10

V[5]
9

V[1]	V[2]	V[3]	V[4]	V[5]
3	7	8	9	10

V[6]
12

V[1]	V[2]	V[3]	V[4]	V[5]	V[6]
3	7	8	9	10	12

Algoritmul în C++ este următorul:

```
void inserare_directa(int n, int (&v)[20])
{int j,element,i;
   for(j=2;j<=n;j++)
       {
       i=j-1;
        element=v[j];
        while(i>=1 && v[i]>element)
              {v[i+1]=v[i];
              i=i-1;}
       v[i+1]=element;}
       }
```

ANALIZA SORTĂRII PRIN INSERARE

Vom relua prezentarea subprogramului sortarea prin inserţie, adăugând costul de timp pentru fiecare instrucţiune şi a numărului care reprezintă de câte ori acesta este efectiv executată. Pentru fiecare *j=2, 3, ..., n* unde *n* este lungimea vectorului, vom nota t_j numărul de execuţii ale testului ***cât timp*** din linia 4 pentru valoarea fixată *j*.

```
Subalgoritm inserare_directa (n,A)                          cost   timp
1: pentru j =2,n executa                                    c1     n
2:     element = v[j]                                       c2     n-1
3:     i = j -1                                             c3     n-1
4:     cât timp((i>=1) si (v[i]>element)) execută           c4     Σ_{j=2}^{n} t_j
5:            v[i+1] = v[i]                                 c5
       Σ_{j=2}^{n}(t_j − 1)
6:            i = i-1                                       c6
       Σ_{j=2}^{n}(t_j − 1)
       sfârşit cât timp
7:     v[i+1] = element                                     c8     n-1
```

Timpul de execuţie al algoritmului este suma tuturor timpilor de execuţie corespunzătoare fiecărei instrucţiuni executate. O instrucţiune care consuma timpul c_i pentru execuţie şi este executată de ***n*** ori, va contribui cu $c_i * n$ la timpul total de execuţie.

Observaţie

Un fenomen de acest tip nu are loc atunci când se referă la alte resurse, cum ar fi de exemplu, memoria. O instrucţiune care se referă la ***m*** cuvinte de memorie şi este executată de ***n*** ori, nu consumă în general ***m*n*** cuvinte de memorie.

Pentru a calcula *T(n)*, timpul de execuţie pentru sortarea prin inserţie directă, vom aduna produsele mărimilor indicate în coloanele de cost şi timp, obţinând:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j + c_5 \sum_{j=2}^{n} (t_j - 1) + c_6 \sum_{j=2}^{n} (t_j - 1) + c_7(n-1)$$

Pentru sortarea prin inserţie directă, ***cazul cel mai favorabil*** apare când vectorul de intrare este deja sortat. Pentru fiecare *j=2, 3, ..., n,* vom găsi ***v[i]≤element*** în linia 4, când ***i*** are valoarea iniţială ***j-1***. Rezultă că $t_j=1$ pentru *j=2, 3, ..., n* şi timpul de execuţie în cazul cel mai favorabil este:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1)$$

$$T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)$$

Timp de execuţie poate fi exprimat sub forma ***an+b*** pentru anumite constante ***a*** şi ***b*** care depind doar de timpii de execuţie c_i fiind astfel o ***funcţie liniară*** de ***n***.

Cazul cel mai defavorabil apare dacă vectorul este sortat în ordine inversă – în ordine descrescătoare. În această situaţie trebuie să comparăm fiecare element ***v[j]*** cu fiecare element din subvectorul ***v[1], v[2], …, v[j-1]***, astfel $t_j=j$ pentru ***j=2, 3, …, n***.

Observăm că:

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

şi

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

În cazul cel mai defavorabil timpii de execuţie pentru sortarea prin inserţie directă este:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n+1)}{2}\right) + c_6\left(\frac{n(n+1)}{2}\right) + c_7(n-1)$$

$$T(n) = \left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right)n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7\right)n - (c_2 + c_3 + c_4 + c_7)$$

Timpul de execuţie în **cazul cel mai defavorabil** poate fi exprimat sub forma $an^2 + bn + c$, unde a,b şi c depind de costurile c_i ale instrucţiunilor, fiind astfel o ***funcţie pătratică de ordin n***.

Să presupunem că alegem la întâmplare ***n*** numere şi aplicăm sortarea prin inserţie. Cât timp va fi necesar pentru a determina locul în care putem insera ***v[j]*** în subvectorul ***v[1], v[2], …, v[j-1]***? În medie, jumătate din elementele subvectorului ***v[1], v[2], …, v[j-1]*** sunt mai mici decât ***v[j]***, şi cealaltă jumătate sunt mai mari. Prin urmare, în medie, trebuie verificate jumătate din elementele subvectorului ***v[1], v[2], …, v[j-1]***, deci $t_j = j/2$. Dacă ţinem seama de această observaţie, ***timpul de execuţie mediu*** va apărea tot ca o ***funcţie pătratică de n***, la fel ca în cazul cel mai defavorabil.

Vom considera doar termenul dominant al formulei (adică an^2) deoarece ceilalţi termeni sunt relativ nesemnificativi pentru valori mari ale lui ***n***. Ignorăm, de asemenea şi factorul constant a, deoarece pentru numere foarte mari, factorii constanţi sunt mai puţin semnificativi decât ordinul de creştere în determinarea eficienţei computaţionale a unor algoritmi.

Sortarea prin inserţie are un timp de execuţie în cazul cel mai defavorabil şi în cazul mediu de $\theta(n^2)$.

SORTAREA PRIN INSERARE BINARĂ

PREZENTAREA METODEI

Algoritmul de sortare prin inserare poate fi îmbunătăţit pornind de la observaţia că subtabloul din stânga, în care urmează să se efectueze inserarea, este deja ordonată. În acest caz, căutarea locului de inserare se poate face mai rapid utilizând metoda ***căutării binare***. Aceasta

constă în înjumătăţirea repetată a intervalului vizat până la găsirea locului căutat. Metoda de inserţie se numește ***inserare binară***.

De exemplu, la inserarea celui de-al *32*-lea element putem începe prin a compara v_{32} cu v_{16}; dacă valoare este mai mică, o comparăm cu v_8, iar dacă este mai mare o comparăm cu v_{24} etc., deci locul în care trebuie inserat v_{32} va fi cunoscută după efectuarea a doar şase comparaţii.

Metoda inserţie binară a fost menționată de John Mauchly, în 1946, în prima discuţie publică referitoare la sortarea pe calculator.

PREZENTAREA ALGORITMULUI

Fie vectorul $v_1, v_2, ..., v_n$. Pornim cu primul element v_1 şi la primul pas căutăm, folosind căutarea binară, poziţia în care ar trebui să se găsească v_2. Dacă $v_2 < v_1$, atunci v_2 trebuie să fie înainte de v_1 deci îl mutăm pe v_1, în locul lui v_2.

La pasul i, unde $2 \le i \le n$, avem elementele $v_1, v_2,..., v_{i-1}$ sortare crescător şi încercăm să inserăm elementul v_i. Căutăm poziţia pe care ar trebui inserat elementul v_i folosind căutarea binară. Odată găsită poziţia i pe care va trebui să se plaseze elementul v_i, deplasăm elementelor de la poziţia i, cu câte o poziţie spre dreapta.

Algoritmul în pseudocod este următorul:

```
Subalgoritm inserare_binara (n,v)
  pentru i =2,n executa
     element = v[i]
     inceput = 1
     sfarsit = i-1
     cât timp(inceput<=sfarsit) execută
          mijloc = (inceput+sfarsit)/2
          dacă (element < v[mijloc])
               sfarsit=mijloc-1
          altfel
               inceput=mijloc+1
      sfârşit cât timp
     pentru j=i,inceput+1,-1 execută
          v[j] = v[j-1]
      sfârşit pentru
     v[inceput]=element
sfârşit pentru
sfârşit subalgoritm
```

Exemplu

Fie următorul vector:

V[1]	V[2]	V[3]	V[4]	V[5]	V[6]
219	128	243	98	491	151

Elementul ce urmează a fi inserat şi comparațiile acestuia cu elementele tabloului	**Ordinea elementelor după inserare**
• **i=1**	V[1]: 219
• **i=2, *mijloc=1*, *element=128***; • se compară ***element*** cu ***v[1]*** • se inserează ***element*** pe **poziția 1**	V[1]: 128 V[2]: 219
• **i=3**, ***mijloc=1***, ***element=243***; • se compară ***element*** cu ***v[1]*** • ***mijloc=2*** • se compară ***element*** cu ***v[2]*** • se inserează ***element*** pe **poziția 3**	V[1]: 128 V[2]: 219 V[3]: 243
• **i=4**, ***mijloc=2, element=98***; • se compară ***element*** cu ***v[2]*** • ***mijloc=1***; • se compară ***element*** cu ***v[1]*** • se inserează ***element*** pe **poziția 1**	V[1]: 98 V[2]: 128 V[3]: 219 V[4]: 243
• **i=5**, ***mijloc=2, element=491*** • se compară ***element*** cu ***v[2]*** • ***mijloc=3***; • se compară ***element*** cu ***v[3]*** • ***mijloc=4*** • se compară ***element*** cu ***v[4]*** • se inserează ***element*** pe **poziția 5**	V[1]: 98 V[2]: 128 V[3]: 219 V[4]: 243 V[5]: 491
• **i=6**, ***mijloc=3***, ***element=151*** • se compară ***element*** cu ***v[3]*** • ***mijloc=1*** • se compară ***element*** cu ***v[1]*** • ***mijloc=2*** • se compară element cu ***v[2]*** • se inserează ***element*** pe **poziția 3**	V[1]: 98 V[2]: 128 V[3]: 151 V[4]: 219 V[5]: 243 V[6]: 491

Algoritmul în C++ este următorul:

```
void inserare_binara(int n, int (&v)[20])
{int j,element,i,inceput,sfarsit,mijloc;
  for(i=2;i<=n;i++)
  {
  element=v[i];
  inceput=1;
  sfarsit=i-1;
  while(inceput<=sfarsit)
   {mijloc=(inceput+sfarsit)/2;
    if(element<v[mijloc])
      sfarsit=mijloc-1;
    else
      inceput=mijloc+1;
       }
  for(j=i;j>=inceput+1;j--)
     v[j]=v[j-1];
  v[inceput]=element;
 }
}
```

COMPLEXITATEA ALGORITMLUI

Conform analizei privind căutarea binară, pentru a determina locul în care trebuie inserat v_k între celelalte înregistrări astfel încât $v_1 \leq v_2 \leq \ldots \leq v_{k-1}$ sunt necesare cel mai mult $[\log_2 k]$ comparaţii. În aceste condiţii, metoda sortării prin inserţie binară necesită cel mult T(n) = $\sum_{k=1}^{n} [\log_2 k]$ comparaţii. Se poate arăta că T(n)= $n[\log_2 n] - 2^{[\log_2 n]} + 1$.

ÎMBUNĂTĂŢIREA SORTĂRII PRIN INSERARE BINARĂ

SORTAREA PRIN INSERŢIE ÎN DUBLU SENS

Din nefericire, inserţia binară, rezolvă numai jumătate din problemă. După găsirea locului unde trebuie inserat elementul *v[j]*, va trebui să deplasăm aproximativ $\frac{1}{2} j$ din elementele sortate anterior pentru a face loc lui *v[j]*, astfel că timpul total de rulare va rămâne proporţional cu n^2.

Prima încercare propusă la începutul anilor 1950 se numeşte ***inserţie în dublu sens***. Primul element este pus în centrul unei zone de ieşire, iar pentru elementele ce urmează se face spaţiu prin deplasare la stânga sau la dreapta, după cum este mai convenabil. Aceasta va economisi aproximativ jumătate din timpul de rulare faţă de inserţia binară obţinută. Preţul plătit pentru această economie constă într-un program ceva mai complicat. Această metodă se poate utiliza fără a solicita spaţiu mai mult decât pentru cele n elemente.

Fie un următorul tablou:

V[1]	V[2]	V[3]	V[4]	V[5]	V[6]
219	128	243	98	491	151

Inserarea cu deplasarea în dublu sens este reprezentată în următorul tabel:

V[1]
219

V[1]	V[2]
128	219

V[1]	V[2]	V[3]
128	219	243

V[1]	V[2]	V[3]	V[4]
98	128	219	243

V[1]	V[2]	V[3]	V[4]	V[5]
98	128	219	243	491

V[1]	V[2]	V[3]	V[4]	V[5]	V[6]
98	128	151	219	243	491

SORTARE PRIN METODA LUI SHELL

Un algoritm de sortare care deplasează elementele numai cu câte o poziţie deodată va avea un timp mediu de rulare cu n^2, în cel mai bun caz, deoarece fiecare element trebuie deplasat în medie aproximativ $\frac{1}{3}n$ poziţii, în timpul procesului de sortare. Pentru a îmbunătăţi inserţia directă, avem nevoie de un mecanism, prin intermediul căruia elementele să efectueze salturi mari în loc de paşi mici.

O astfel de metodă a fost propusă de către Donald L. Shell în 1959 şi este o completare a algoritmului de inserţie directă. El mai poartă numele şi de ***sortare prin micșorarea incrementului***.

Elementul cheie al algoritmului îl reprezintă alegerea pasului h.

PREZENTAREA METODEI

Considerăm un vector 16 elemente $v_1, v_2, ..., v_{16}$. Aceste elemente le vom împărţi în 8 grupe a câte două, adică (v_1, v_9), (v_2, v_{10}), ... (v_8, v_{16}). Sortând fiecare grupă separat, ne va conduce la următorul pas al sortării. Acum, vom împărţi elementele în câte 4 grupe de câte patru înregistrări fiecare, adică (v_1, v_5, v_9, v_{16}), ..., $(v_4, v_8, v_{12}, v_{16})$, şi din nou fiecare grup este sortat separat. A treia trecere va sorta două grupe de opt elemente, apoi o a patra trecere termină treaba sortând toate cele 16 înregistrări.

Prin această metodă elementele tind să conveargă rapid spre destinaţia lor.

În exemplu dat, s-a utilizat secvenţa de incremente 8, 4, 2, 1. Dar poate fi utilizată orice secvenţă $h_t, h_{t-1}, ...,h_1$, cu condiţia ca ultimul increment h, să fie egal cu 1.

În program s-a utilizat împărţirea şirului considerând $h_{i+1}=h_i*3+1$.

O altă variantă de alegere a pasului h, pentru care s-a demonstrat că are complexitatea $O(n\sqrt{n})$ este $h_k=2^k-1$.

PREZENTAREA ALGORITMULUI

Elementele vectorul $v_1, v_2, ..., v_n$, sunt rearanjate pe loc. După terminarea sortării, elementele vor fi în ordinea $v_1 \leq v_2 \leq .... \leq v_n$. Pentru controlul procesului de sortare se utilizează incrementurile $h_t, h_{t-1},...,h_1$, unde $h_1 =1$. Alegerea corectă a incrementurilor poate micşora semnificativ timpul de sortare. Acest algoritm se reduce la sortarea prin inserţie directă când t= 1.

Algoritmul în pseudocod este următorul:

```
Subalgoritm shell (n,v)
  h=1
  cat timp (h<=n)
    h=3*h+1
  sfarsit cat timp
  cat timp (h>0)
    h=h/3
    pentru i=h+1,n executa
          aux=v[i]
          j=i-h
          cat timp (j>=1 && aux<v[j])
               v[j+h]=v[j]
               j=j-h
          sfarsit cat timp
     v[j+h]=aux
   sfarsit pentru
   sfarsit cat timp
sfârşit subalgoritm
```

Exemplu

Exemplificăm ideea algoritmului în cazul unui şir cu 15 elemente pentru următoarele valori ale pasului h=13, h=4, h=1 care corespunde unui şir h_k dat prin relaţia $h_k=3*h_{k-1}+1$, $h_1=1$.

Etapa 1: pentru **h=13** se aplică algoritmul sortării prin inserţie subşirurilor v[1], v[14] şi v[2], v[15]

V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]	V[8]	V[9]	V[10]	V[11]	V[12]	V[13]	V[14]	V[15]
14	8	10	7	9	12	5	15	2	13	6	1	4	3	11

şi se obţine:

V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]	V[8]	V[9]	V[10]	V[11]	V[12]	V[13]	V[14]	V[15]
3	8	10	7	9	12	5	15	2	13	6	1	4	14	11

Etapa 2: pentru **h=4** se aplică algoritmul sortării prin inserţie succesiv subşirurilor:

- v[1], v[5], v[9], v[13];
- v[2], v[6], v[10], v[14];
- v[3], v[7], v[11], v[15];
- v[4], v[8], v[12];

După ***prima subetapă*** (prelucrarea primului subşir) prin care se ordonează subşirul constituit din elementele marcate:

V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]	V[8]	V[9]	V[10]	V[11]	V[12]	V[13]	V[14]	V[15]
3	8	10	7	9	12	5	15	2	13	6	1	4	14	11

se obţine:

V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]	V[8]	V[9]	V[10]	V[11]	V[12]	V[13]	V[14]	V[15]
2	8	10	7	3	12	5	15	4	13	6	1	9	14	11

La ***a doua subetapă*** se aplică sortarea prin inserţie asupra subşirului construit din elementele marcate:

V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]	V[8]	V[9]	V[10]	V[11]	V[12]	V[13]	V[14]	V[15]
2	8	10	7	3	12	5	15	4	13	6	1	9	14	11

se obţine aceeaşi configuraţie (subşirul este ordonat crescător):

V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]	V[8]	V[9]	V[10]	V[11]	V[12]	V[13]	V[14]	V[15]
2	8	10	7	3	12	5	15	4	13	6	1	9	14	11

La ***a treia subetapă*** se aplică sortarea prin inserţie asupra subşirului construit din elementele marcate:

V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]	V[8]	V[9]	V[10]	V[11]	V[12]	V[13]	V[14]	V[15]
2	8	10	7	3	12	5	15	4	13	6	1	9	14	11

se obţine:

V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]	V[8]	V[9]	V[10]	V[11]	V[12]	V[13]	V[14]	V[15]
2	8	5	7	3	12	6	15	4	13	10	1	9	14	11

Se aplică sortarea prin inserţie asupra subşirului constituit din elementele marcate:

V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]	V[8]	V[9]	V[10]	V[11]	V[12]	V[13]	V[14]	V[15]
2	8	5	7	3	12	6	15	4	13	10	1	9	14	11

se obţine:

V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]	V[8]	V[9]	V[10]	V[11]	V[12]	V[13]	V[14]	V[15]
2	8	5	1	3	12	6	7	4	13	10	15	9	14	11

Etapa 3: pentru **h=1** se aplică algoritmul sortării prin inserţie în urmă căruia se obţine şirul ordonat:

V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]	V[8]	V[9]	V[10]	V[11]	V[12]	V[13]	V[14]	V[15]
2	8	5	1	3	12	6	7	4	13	10	15	9	14	11

Se va obţine:

V[1]	V[2]	V[3]	V[4]	V[5]	V[6]	V[7]	V[8]	V[9]	V[10]	V[11]	V[12]	V[13]	V[14]	V[15]
1	2	3	4	5	5	7	8	9	10	11	12	13	14	15

Algoritmul în C++ este următorul:

```
void shell(int n, int (&v)[25])
{int h,i,j,aux;
 h=1;
 while (h<=n)
   h=3*h+1;
 while (h>0)
{h=h/3;
 for (i=h+1;i<=n;i++)
  {aux=v[i];
   j=i-h;
   while (j>=1 &&aux<v[j])
     {v[j+h]=v[j];
     j=j-h;}
     v[j+h]=aux;
     }
   }
}
```

Sortările h-sortat se fac de obicei folosind metoda de sortare prin inserţie directe. Dacă în locul acesteia se foloseşte metoda bulelor, metoda obţinută se numeşte ***Brick Sort***.

Sortarea prin metoda Shell este cel mai rapid algoritm din clasa celor de complexitate secvenţială $O(n^2)$. Viteza algoritmului este aproximativ de cinci ori mai mare decât sortare prin metoda bulelor şi de peste două ori mai mare decât metoda de sortare prin inserţie.

BIBLIOGRAFIE:

1. Donald E. Knuth – Arta programării calculatoarelor, Volumul III , Editura Teora, 2001
2. Thomas H. Cormen, Charles E. Leiserson, Ronald R. Rivest – Introducere în algoritmi, Editura Agora Cluj, 2008.
3. http://web.info.uvt.ro/~dzaharie/alg/alg2011_cap4.pdf