Sortări simple

CAPITOLUL V

Cuprins

Introducere

Prin sortare înțelegem ordonarea unei colecții de elemente după un criteriu dat, rezultatul fiind o permutare a datelor de intrare.

Sortăm diferite lucruri în viața de zi cu zi: ex. cărțile din bibliotecă, listele de studenți, rezultatele la un campionat, etc.

În cele mai multe cazuri **datele pot fi mai ușor gestionate** dacă sunt sortate după un anumit criteriu => nevoia dezvoltării unor algoritmi/metode de sortare.

Sortarea este **una dintre cele mai frecvente sarcini** în domeniul calculatoarelor => un interes deosebit acordat tehnicilor de sortare.

De ce este importantă studierea algoritmilor de sortare?

Introducere

De ce este **importantă** studierea algoritmilor de sortare?

- Pentru a putea **compara** diferiți algoritmi și pentru a alege algoritmul cel mai potrivit pentru aplicația noastră
- Pentru a **înțelege** diverse variante ale tehnicii de programare **Divide et Impera** (Devide and Conquer)
- Pentru a **ilustra** o serie de **metode de analiză** a algoritmilor
- Pentru a **exemplifica metode de îmbunătățire** a performanțelor unui algoritm
- Sortarea este domeniul ideal al studiului **construcției algoritmilor**, **performanțelor algoritmilor** și al **tehnicilor de programare**

Introducere

Cum **analizăm** algoritmii de sortare?

- Criteriile relevante pentru compararea unor algoritmi de sortare sunt **numărul de comparații** și **numărul de interschimbări**, care nu coincid în mod obligatoriu.
- După cum am vazut în capitolele anteriore, să determinăm numărul exact de comparații ce se efectuează asupra datelor în procesul de sortare este dificil de realizat, chiar imposibil, de aceea se recurge la **aproximări asimptotice** de tipul $\Theta(f(n))$ și/sau $O(f(n))$.
- Pentru algoritmii de sortare se aproximează asimptotic pentru **cazul cel mai favorabil, cel mai defavorabil** și **cazul mediu**.
- Alegerea unui algoritm se va face și ținând cont de **considerente practice**, nu doar de eficiență.

Terminologie și notații

Prin **sortare** se întelege în general ordonarea unei mulțimi de elemente, cu scopul de **facilita căutarea** ulterioară a unui element dat

În cadrul acestui capitol se presupune că sortarea se referă la anumite elemente care au o structură articol definită după cum urmează:

```
typedef struct {
   int cheie;
   //alte campuri
} tip_element;
```

Terminologie și notații

```
typedef struct {
   int cheie;
   //alte câmpuri
} tip_element;
```

Câmpul **cheie** precizat poate fi neesențial din punct de vedere al informației înregistrate în articol, partea esențială a informației fiind conținută în celelalte câmpuri

Din punct de vedere al sortării, câmpul cheie este cel mai **important** câmp

Tipul câmpului cheie se presupune a fi întreg pentru o mai bună înțelegere, în realitate el putând fi orice tip **scalar**

Terminologie și notații

Pentru tipul *struct* definit anterior, ca de altfel pentru orice elemente care trebuie sortate, avem definiți următorii operatori:

- *in = **Compară** (el1, el2), unde in reprezintă un întreg, si el1, el2 reprezintă două elemente de tipul elemetelor ce trebuie comparate (sortate). Comparația se va face pe baza câmpului cheie.*
- ***Interschimbă** (pel1, pel2), inteschimbă două elemente în memorie, pel1, pel2 specifică locațiile de memorie unde se află cele două elemente ce trebuie interschimbate (funcția este întâlnită și cu numele de swap)*

Terminologie și notații

Pentru in = **Compară** (el1, el2) se vor folosie fie operatori logici (<,>,<=,>=,==) pentru compararea câmpului cheie, fie funcții predefinite (ex. strcmp în cazul șirurilor de caractere)

Implementare În C propusă pentru **Interschimbă** (pel1, pel2):

```
void swap(tip_element *el1, tip_element *el2)
{
   tip_element tmp;
   tmp = *el1;
   *el1 = *el2;
   *el2 = tmp;
}
```

Terminologie și notații

Find dat un set de articole, de tipul structură definit anterior:

$$a_1, a_2, a_3 \dots a_n$$

Prin **sortare** se înțelege permutarea elementelor şirului într-o anumită ordine:

$$a_{k1}, a_{k2}, a_{k3} \dots a_{kn}$$

astfel încât șirul cheilor să devină **monoton crescător**

$$a_{k1}.cheie \leq a_{k2}.cheie \leq a_{k3}.cheie \leq \cdots \leq a_{kn}.cheie$$

Terminologie și notații

Conform definiției anterioare, setul de date de intrare poate conține valori care se repetă

Există și algoritmi de sortare care funcționează exclusiv pe date unice

Pentru cazul în care repetarea valorilor este permisă, valorile duplicate pot fi ordonate după un criteriu implicit

O metodă de sortare se spune că este **stabilă** dacă după sortare, **ordinea relativă** a elementelor cu chei egale coincide cu cea inițială

Această stabilitate este esenţială în special în cazul în care se execută sortarea după mai multe chei

Terminologie și notații

În cazul operaţiei de sortare, **dependenţa** dintre **algoritmul** care realizează sortarea şi **structura de date** prelucrată este profundă.

Din acest motiv **metodele de sortare** sunt clasificate în **două mari categorii** după cum sunt înregistrate elementele de sortat:

- Sunt înregistrate ca şi tablouri în memoria **centrală** a sistemului de calcul, ceea ce conduce la sortarea tablourilor numită sortare **internă**.
- Sunt înregistrate într-o memorie **externă**, ceea ce conduce la sortarea fişierelor (secvenţelor) numită şi sortare **externă**.

Sortarea tablourilor

Tablourile se înregistrează în **memoria centrală** a sistemelor de calcul, motiv pentru care **sortarea tablourilor** se mai numeşte şi **sortare internă**

Cerinţa fundamentală care se formulează faţă de metodele de sortare a tablourilor se referă la utilizarea cât mai economică a zonei de **memorie** disponibile.

Din acest motive pentru început, prezintă interes numai algoritmii care realizează sortarea "**in situ**", adică chiar în zona de memorie alocată tabloului.

Pornind de la această restricţie, în continuare algoritmii vor fi clasificaţi în funcţie de eficienţa lor, respectiv în funcţie de timpul de execuţie pe care îl necesită.

Sortarea tablourilor

Aprecierea **cantitativă** a eficienţei unui algoritm de sortare se realizează prin intermediul unor **indicatori specifici**.

- Un prim indicator este **numărul comparaţiilor de chei** notat cu **C**, pe care le execută algoritmul în vederea sortării.
- Un alt indicator este **numărul de interschimbări de elemente**, respectiv numărul de mişcări de elemente executate de algoritm, notat cu **M**.

Ambii indicatori depind de numărul total *n* al elementelor care trebuiesc sortate.

Sortarea tablourilor

În cazul unor algoritmi de sortare simpli bazaţi pe aşa-zisele **metode directe de sortare** atât C cât şi M sunt proporţionali cu n^2 adică sunt $O(n^2)$.

Există însă şi **metode avansate de sortare**, care au o complexitate mult mai mare şi în cazul cărora indicatorii C şi M sunt de ordinul lui $n * log_2 n$ ($O(n * log_2 n)$).

Raportul $n^2/(n * log_2 n)$, care ilustrează câştigul de eficienţă realizat de aceşti algoritmi, este aproximativ egal cu 10 pentru $n = 64$, respectiv 100 pentru $n =$ 1000.

Sortari simple

Metode de sortare directe

Fac parte din categoria $\Theta\ (n^2)$

Prezintă interes din următoarele motive:

1. Sunt foarte potrivite pentru explicitarea principiilor majore ale sortării.
2. Funcțiile care le implementează sunt scurte şi relativ uşor de înţeles.
3. Deşi metodele avansate necesită mai puţine operaţii, aceste operaţii sunt mult mai complexe în detaliile lor, respectiv metodele directe se dovedesc a fi superioare celor avansate pentru valori mici ale lui n.
4. Reprezintă punctul de pornire pentru metodele de sortare avansate.

Sortari simple

Metodele de sortare care realizează sortarea "**in situ**" se pot clasifica în trei mai categorii:

1. Sortarea prin **inserţie**
2. Sortarea prin **selecţie**
3. Sortarea prin **interschimbare**

Sortarea prin inserție

Sortarea prin inserție sau **insertion sort** este o sortare directă (simplă)

Această metodă este larg utilizată de jucătorii de cărţi

- Elementele (cărţile) sunt în mod conceptual divizate într-o secvenţă **destinaţie** $a_1...a_{i-1}$ şi într-o secvenţă **sursă** $a_i....a_n$
- În fiecare pas, începând cu *i = 1* , elementul *i* al tabloului (în cazul limbajului C în care indecșii încep de la *0),*care este de fapt primul element al secvenţei sursă este luat şi transferat în secvenţa destinaţie prin **inserarea** sa la locul potrivit
- Se incrementează *i* şi se reia ciclul

Sortarea prin inserție

index	i=1	i=2	i=3	i=4	i=5	i=6	i=7	
0	42	20	17	13	13	13	13	13
1	20	42	20	17	17	14	14	14
2	17	17	42	20	20	17	17	15
3	13	13	13	42	28	20	20	17
4	28	28	28	28	42	28	23	20
5	14	14	14	14	14	42	28	23
6	23	23	23	23	23	23	42	28
7	15	15	15	15	15	15	15	42
	tmp=20	tmp=17	tmp=13	tmp=28	tmp=14	tmp=23	tmp=15	

Sortarea prin inserție

index	i=1		i=2		i=3		i=4		i=5		i=6		i=7		
0	42	42	20	20	17	17	13	13	13	13	13	13	13	13	13
1	20	42	42	20	20	17	17	17	17	17	14	14	14	14	14
2	17	17	17	42	42	20	20	20	20	17	17	17	17	17	15
3	13	13	13	13	13	42	42	42	28	20	20	20	20	17	17
4	28	28	28	28	28	28	28	42	42	28	28	28	23	20	20
5	14	14	14	14	14	14	14	14	14	42	42	28	28	23	23
6	23	23	23	23	23	23	23	23	23	23	23	42	42	28	28
7	15	15	15	15	15	15	15	15	15	15	15	15	15	42	42
	tmp=20		tmp=17		tmp=13		tmp=28		tmp=14		tmp=23		tmp=15		

Sortarea prin inserție

La început se sortează primele două elemente, apoi primele trei elemente şi aşa mai departe.

Se face precizarea că în pasul *i*, din exemplul anterior, primele *i* elemente sunt deja sortate, astfel încât sortarea constă numai în a insera elementul *a[i]* la locul potrivit într-o secvenţă deja sortată.

Selectarea locului în care trebuie inserat *a[i]* se face parcurgând **secvenţa destinaţie** deja sortată *a[0],...,a[i-1]* de la dreapta la stânga şi comparând pe *a[i]* cu elementele secvenţei

Simultan cu parcurgerea, se realizează şi **deplasarea spre dreapta** cu o poziţie a fiecărui element testat până în momentul îndeplinirii condiţiei de oprire. În acest mod se face loc în tablou elementului care trebuie inserat.

Oprirea parcurgerii se realizează pe primul element *a[j]* care are cheia mai mică sau egală cu *a[i]*. Dacă un astfel de element *a[j]* nu există, oprirea se realizează pe *a[0]*.

Sortarea prin inserție

Pseudocod:

```
insertion_sort( a[], n)
        pentru i=1; i< n-1; i++
                reține a[i]
                mută toate elementele a[j] mai mari decât a[i] cu o poziție
                copiază a[i] pe poziția potrivită
```

Sortarea prin inserție

O variantă de **implementare** în C:

```
void insertion_sort(tip_element a[], int n)
{
      int i, j;
      tip_element tmp;
      for ( i = 1; i < n; i++)
      {
            tmp = a[i]; /*salvarea elementului de inserat, intr-o
zona de memorie tampon*/
            for ( j = i; (j>0) && (tmp.cheie < a[j-1].cheie); j--)
                  a[j] = a[j - 1];
                   /* mutam elementele din sirul sortat cu cheia mai
mare decat a elementului de inserat spre dreapta */
            a[j] = tmp;        /* inseram elementul la locul potrivit
in sirul sortat */
      }
}
```

Sortarea prin inserție

Analiza sortării prin inserție:

După cum se observă, algoritmul de sortare conţine un **ciclu exterior** după *i* care se reia de *n-1* ori (prima buclă **for)**.

În cadrul fiecărui ciclu exterior se execută un **ciclu interior** de lungime variabilă după *j*, până la îndeplinirea condiţiei (a doua buclă **for**).

În pasul *i* al ciclului exterior **for**, numărul minim de reluări ale ciclului interior este 0 (zero) iar numărul maxim de reluări este *i* .

Sortarea prin inserție

Numărul de **iterații** pentru **bucla exterioară** este fix (n-1)

Numărul de **comparații de chei (C)** pentru **bucla interioară** depinde de numărul de elemente din șirul destinație mai mari decât elementul de inserat, astfel pentru bucla interioară avem:

- O singură comparație pentru datele deja sortate – cazul cel mai **favorabil**
- *i* comparații pentru datele în ordine inversă – cazul cel mai **defavorabil**
- (i+1)/2 în medie, presupunând că toate permutările celor *n* chei sunt egal posibile – cazul **mediu.** Probabilitatea se calculează în felul următor:

$$P = \frac{1 + 2 + 3 + \cdots i}{i} = \frac{\frac{1}{2} i(i+1)}{i} = \frac{i+1}{2}$$

Sortarea prin inserție

Numărul **total** de **comparații de chei**(**C**):

- Cazul cel mai favorabil:

$$C_{min} = \sum_{i=1}^{n-1} \mathbf{1} = n - 1 = O(n)$$

- Cazul cel mai defavorabil:

$$C_{max} = \sum_{i=1}^{n-1} \boldsymbol{i} = 1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2} = O(n^2)$$

- Cazul mediu:

$$C_{me} = \sum_{i=1}^{n-1} \frac{\boldsymbol{i} + \mathbf{1}}{\mathbf{2}} = \sum_{i=1}^{n-1} \frac{\boldsymbol{i}}{\mathbf{2}} + \sum_{i=1}^{n-1} \frac{1}{2} = \frac{\frac{1}{2} n(n-1)}{4} + \frac{1}{2}(n-1) = \frac{n^2 + n - 2}{4} = O(n^2)$$

Sortarea prin inserție

Numărul de **atribuiri de elemente M** pentru **bucla interioară** este egal cu numărul de comparații de chei din acea buclă (notat C_i), deoarece în interiorul buclei interioare avem doar o atribuire

Numărul de **atribuiri M** pentru o iterație a **buclei exterioare** este egal cu C_i +2 (numărul de atribuiri din bucla interioară tmp = a[i]; și a[j]=tmp), atfel numărul total M este:

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

Sortarea prin inserție

Numărul **total** de **atribuiri de elemente**(M):

- Cazul cel mai favorabil:

$$M_{min} = \sum_{i=1}^{n-1} (\mathbf{1} + \mathbf{2}) = 3(n-1) = O(n)$$

- Cazul cel mai defavorabil:

$$M_{max} = \sum_{i=1}^{n-1} (\boldsymbol{i} + \mathbf{2}) = \frac{n(n-1)}{2} + 2(n-1) = \frac{n^2 + 3n - 4}{2} = O(n^2)$$

- Cazul mediu:

$$M_{med} = \sum_{i=1}^{n-1} (\frac{\boldsymbol{i} + \mathbf{1}}{\mathbf{2}} + \mathbf{2}) = \frac{\frac{1}{2}n(n-1)}{4} + \frac{3}{2}(n-1) = \frac{n^2 + 5n - 6}{4} = O(n^2)$$

Sortarea prin inserție

Avantaje:

- O sortare simplă
- O sortare stabilă
- Necesită puțin spațiu de memorie adițional, iar acesta nu depinde de dimensiunea setului de date de intrare (O(1))
- Avantajoasă pentru seturi de date sortate parțial (vezi cazul cel mai favorabil)
- Nu necesită cunoașterea tuturor elementelor în avans => poate fi adaptată pentru situații de programare dinamică (pentru date care se primesc secvențial)

Sortarea prin inserție

Dezavantaje:

- Ineficientă pentru cazul cel mai defavorabil și mediu (atât **C** cât și **M** sunt $O(n^2)$)
- În general, nu este o metodă potrivită de sortare cu ajutorul calculatorului, deoarece inserţia unui element presupune deplasarea poziţie cu poziţie în tablou a unui număr de elemente, deplasare care este neeconomică

Sortarea prin selecție

Sortarea prin selecție sau **selection sort** este tot o sortare directă (simplă) ca și sortarea prin inserție

Asemenea sortării prin inserție și sortarea prin selecție împarte tabloul într-un șir sursă (nesortat) și unul destinație (sortat)

Sortarea prin selecţie foloseşte procedeul de a **selecta** elementul cu cheia minimă din șirul sursă şi de a schimba între ele poziţia acestui element cu cea a elementului următor șirului destinație.

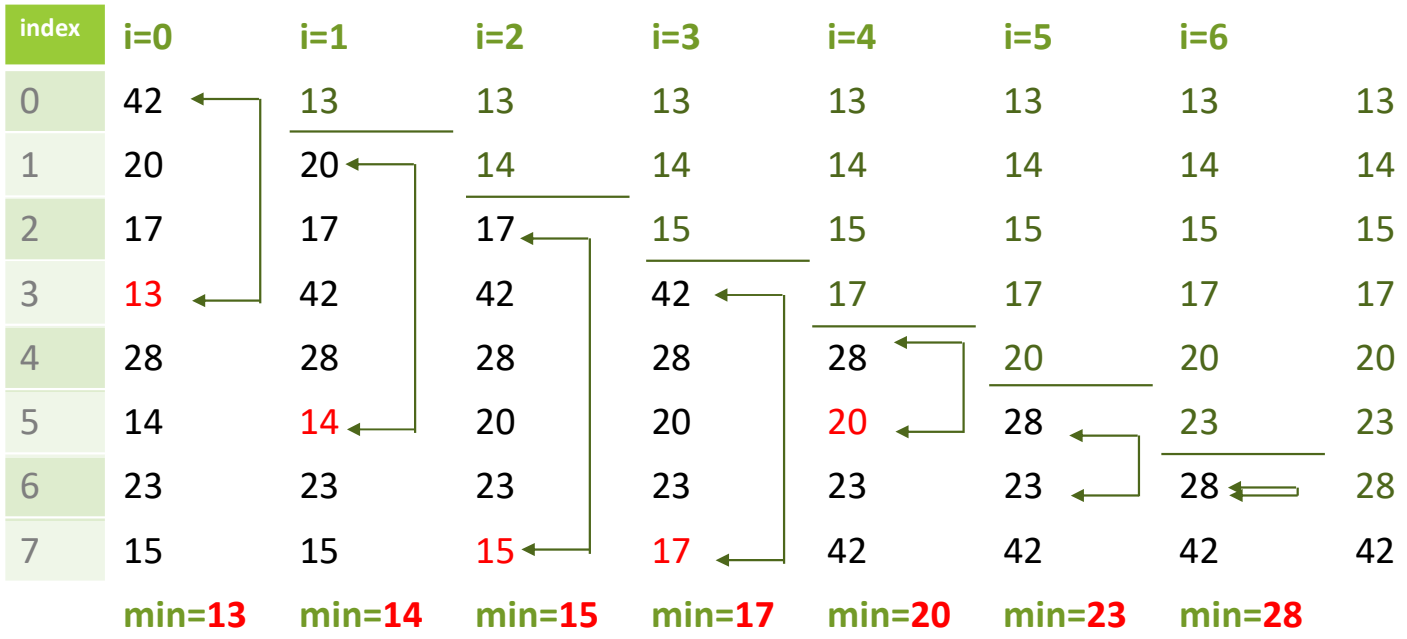
Se repetă acest procedeu cu cele *n*-1 elemente rămase, apoi cu cele *n*-2, etc. terminând cu ultimele două elemente.

Sortarea prin selecție

Această metodă este oarecum opusă **sortării prin inserţie** care consideră la fiecare pas **un singur element al secvenţei sursă** şi **toate elementele secvenţei destinaţie** în care se caută de fapt locul de inserţie.

Selecţia în schimb presupune **toate elementele secvenţei sursă** dintre care îl selectează pe cel cu cheia cea mai mică şi îl depozitează ca **element următor al secvenţei destinaţie.**

Sortarea prin selecție



Sortarea prin selecție

Pseudocod:

```
selection_sort( a[], n)
        pentru i=0; i< n-1; i++
                reține elementul cu valoarea minima din șirul a[i],…, a[n-1]
                interschimba elementul cu valoarea minima cu a[i]
```

Sortarea prin selecție

O variantă de **implementare** în C:

```
void selection_sort(tip_element a[], int n)
{
    int i, j, min; /* min retine INDEXUL
    elementului cu valoare minima */
    for (i = 0; i < n - 1; i++)
    {
        min = i;
        for (j = i + 1; j < n; j++)  /* cautam
        minimul in sirul sursa */
        {
            if (a[j].cheie < a[min].cheie)
            min = j;
        }
        swap(&a[min], &a[i]);  /* interschimba
        cele doua elemente */
    }
}
```

```
void swap(tip_element
*el1, tip_element *el2)
{
    tip_element tmp;
    tmp = *el1;
    *el1 = *el2;
    *el2 = tmp;
}
```

Sortarea prin selecție

Analiza sortării prin selecție:

Ca în cazul sortării prin inserție avem tot două cicluri (for)

Ciclul exterior se execută de n-1 ori

Ciclul interior se execută de n-1-i ori

Comparațiile de elemente (C) se execută doar în ciclul interior

C este același pentru **toate cazurile**:

$$C = \sum_{i=0}^{n-2} (n-1-i) = (n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2} = O(n^2)$$

Sortarea prin selecție

Pentru varianta de implementare propusă anterior, avem și numărul de interschimbări constant.

Avem câte 3 atribuiri (date de interschimbarea elementelor-swap) pentru fiecare valoare a lui *i*

$$M = 3(n-1)$$

Cazul cel mai favorabil este atins când elementele sunt deja sortate

Se observă dezavantajul faptului că în ciuda faptului că tabloul este sortat, se fac totuși interschimbări (elementul cu el însuși, când indecșii *min* și *i* au aceeași valoare).

M_{min} poate fi îmbunătățit prin adăugarea unei verificări suplimentare, dar în acest caz introducem *n-1* comparații (comparații de indecși în acest caz, nu de elemente, neinfluențând valoarea lui C).

```
if( i != min)
        swap(&a[min], &a[i]);  //interschimba cele doua elemente doar daca sunt diferite
```

Sortarea prin selecție

Avantaje:

- O sortare simplă
- Necesită puțin spațiu de memorie adițional, iar acesta nu depinde de dimensiunea setului de date de intrare (O(1))
- Eficientă din punct de vedere al interschimbărilor

Sortarea prin selecție

Dezavantaje:

- Nu este o sortare stabilă
- Nu este eficientă din punct de vedere al comparațiilor ($O(n^2)$)

Sortarea prin interschimbare

Sortarea prin interschimbare sau **bubble sort** este tot o sortare simplă directă

Este una din primele metode de sortare învățate în cursurile de programare de bază

Ne imaginăm tabloul de sortat ca o coloană verticală, cu indexul 0 în partea ce mai de sus

Tabloul se parcurge de jos în sus (de la indecșii superiori spre cei inferiori) și de câte ori întâlnim două elemente adiacente care nu sunt în odinea potrivită unul față de altul, acestea se interschimbă

În acest mod elementele ajung pas cu pas la locul potrivit, avansând de jos în sus asemenea unor bule de gaz intr-un lichid, de unde și numele de bubble sort (din limba engleză)

Pentru acest algoritm interschimbarea elementelor două câte două este caracteristica dominantă, de unde și numele de sortare prin interschimbare din limba română

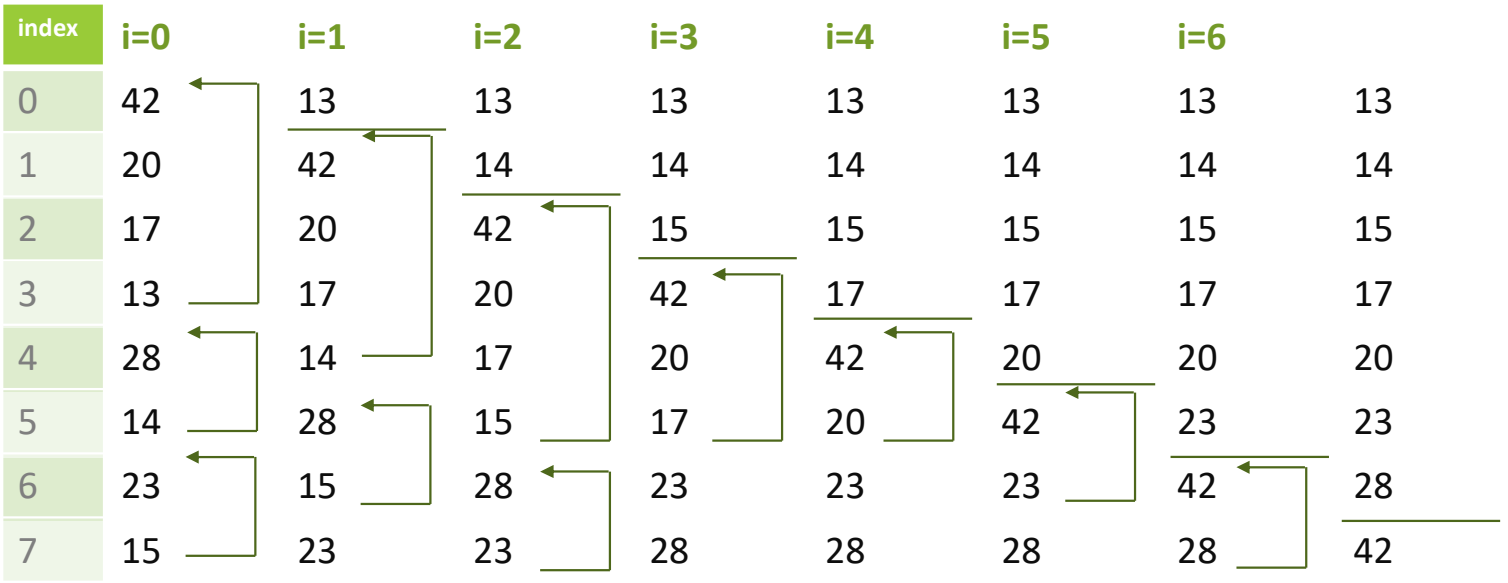
Sortarea prin interschimbare

Principiul de bază al **sortării prin interschimbare** este următorul:

Se compară şi se interschimbă perechile de elemente alăturate, până când toate elementele sunt sortate.

Ca şi la celelalte metode, se realizează **treceri repetate** prin tablou, de la capăt spre început, de fiecare dată deplasând cel mai mic element al mulţimii rămase spre capătul din stânga al tabloului.

Sortarea prin interschimbare



index	i=0	i=1	i=2	i=3	i=4	i=5	i=6	
0	42	13	13	13	13	13	13	13
1	20	42	14	14	14	14	14	14
2	17	20	42	15	15	15	15	15
3	13	17	20	42	17	17	17	17
4	28	14	17	20	42	20	20	20
5	14	28	15	17	20	42	23	23
6	23	15	28	23	23	23	42	28
7	15	23	23	28	28	28	28	42

Sortarea prin interschimbare

Pseudocod:

```
bubble_sort( a[], n)
        pentru i = 0; i < n-1; i++
                pentru j = n-1; j > i; j--
                interschimbă elementele de la pozițiile i și j dacă nu sunt în ordinea potrivită
```

Sortarea prin interschimbare

Propunere implementare **C**:

```
void bubble_sort(tip_element a[], int n)
{
   int i, j;
   for (i = 0; i < n - 1; i++)
      for (j = n - 1; j>i;j--)
            if (a[j].cheie < a[j - 1].cheie)
            /* daca elementele nu sunt in
      ordinea potrivita */
                  swap(&a[j], &a[j - 1]);
}
```

```
void swap(tip_element
*el1, tip_element *el2)
{
   tip_element tmp;
   tmp = *el1;
   *el1 = *el2;
   *el2 = tmp;
}
```

Sortarea prin interschimbare

Analiza sortării prin interschimbare:

Ca în cazul sortării prin inserție și selecție avem tot două cicluri (for)

Ciclul exterior se execută de n-1 ori

Ciclul interior se execută de n-1-i ori

Comparațiile de elemente (C) se execută doar în ciclul interior

C este același pentru **toate cazurile**:

$$C = \sum_{i=0}^{n-2} (n-1-i) = (n-1) + (n-2) + \cdots + 1 = \frac{n(n-1)}{2} = O(n^2)$$

Sortarea prin interschimbare

Pentru calculul numărului de atribuiri de elemente avem pentru **cazul cel mai defavorabil**, câte 3 (date de funcția swap) pentru fiecare iterație a ciclului interior:

$$M_{max} = 3 * C == \frac{3n(n-1)}{2} = O(n^2)$$

Pentru cazul cel mai favorabil, când elementele sunt deja sortate nu avem atribuiri de elemente:

$$M_{mi} = 0$$

Sortarea prin interschimbare

Pentru cazul mediu, avem următoarea probabilitate

$$P = \frac{1 + 2 + 3 + \cdots n - 1 - i}{n - i} = \frac{n - i - 1}{2}$$

Numărul de mutări mediu devine:

$$M_{med} = 3 * \sum_{i=0}^{n-2} \frac{(\mathrm{n} - 1 - \mathrm{i})}{2} = \frac{3n(n-1)}{4} = O(n^2)$$

Sortarea prin interschimbare

Avantaje

- Ușor de implementat

Dezavantaje

- Pentru cazul mediu sortarea prin interschimbare face de doua ori mai multe comparații față de sortarea prin inserție și același număr de atribuiri de elemente
- Pentru cazul mediu sortarea prin interschimare face același număr de comparații cu sortarea prin selecție, dar de n ori mai multe atribuiri de elemente

Concluzii

O comparație a algorimilor prezentați (în variantele de implementare propuse)

Număr de **comparații de elemente (C)**

Sortare prin/ Valoare	Insertie	Selectie	Interschimbare
Min	O(n)	$O(n^2)$	$O(n^2)$
Med	$O(n^2)$	$O(n^2)$	$O(n^2)$
Max	$O(n^2)$	$O(n^2)$	$O(n^2)$

Concluzii

O comparație a algorimilor prezentați (în variantele de implementare propuse)

Număr de **atribuiri de elemente (M)**

Sortare prin/ Valoare	Insertie	Selectie	Interschimbare
Min	O(n)	O(n)	0
Med	$O(n^2)$	O(n)	$O(n^2)$
Max	$O(n^2)$	O(n)	$O(n^2)$

Concluzii

O comparație a algorimilor prezentați (în variantele de implementare propuse)

Sortare prin/ Valoare	Insertie	Selectie	Interschimbare
Memorie auxiliară	O(1)	O(1)	O(1)
Sortare stabilă	Da	Depinde de implementare	Da
Observații	Poate fi folosită online		Cod simplu

Exerciții

Ex1: Să se determine eficiența în termeni de O(f(n)) pentru sortarea prin inserție în cazul în care toate elementele din setul de date de intrare au aceeași valoare.

Ex2: Modificați algoritmul pentru sortarea prin inserție astfel încât sortarea să se facă de la sfârșitul tabloului spre început. Ordinea va rămâne tot crescătoare.

Ex3: Scrieți un program bazat pe algoritmul de sortare prin inserție, care să primească ca input o stivă (și nu un tablou unidimensional, ca în exemplul din curs). Algoritmul poate folosi un număr finit de stive și de variabile auxiliare (nu poate folosi tablouri) și trebuie să fie $O(n^2)$ în cazul cel mai defavorabil.

Bibliografie selectivă

- Drozdek, A. (2012). *Data Structures and algorithms in C++*. Cengage Learning.
- Shaffer, C. A. (2012). Data structures and algorithm analysis.
- Crețu, V. Structuri de date și algorimi, Editura Orizonturi Universitare Timișoara, 2011