3. Sortarea

3.1 Conceptul de sortare

- Obiectivele fundamentale ale acestui capitol sunt:
    - Prezentarea unui **set de exemple** care să ilustreze folosirea structurilor de date descrise în capitolul precedent
    - Să prezinte cum **alegerea structurii influențează** datele privind:
        - **Algoritmul** folosit
        - **Tehnica de programare** care implementează algoritmul
- **Sortarea** este domeniul ideal pentru studiul:
    - (1) **dezvoltarea algoritmului**
    - (2) **performanța algoritmului**
    - (3) **avantajele și dezavanatajele** unei aplicații particular ale algoritmului
    - (4) **tehnicile de programare** specific fiecărui algoritm
- **Sprtarea** este înțelesă în general ca procesul de **rearanjare** a unui set de obiecte în **ordine specifică**
    - Scopul sortării este de a facilita **căutările ulterioare** în setul de date
    - **Sortarea este o activitate fundamental**
    - Obiectele sunt sortate în agendele telefonice, în coprinsuri, în biblioteci, în dicționare, în depozite, și aproape oriunde obiectele vor fi căutate și regăsite.
    - Chiar și copii mici sunt învățați să ”își ordoneze” lucrurile, și se cpnfruntă cu acest lucru cu multă vreme înainte să învețe aritmetica.
- În acest capitol, presupunem că sortarea se face cu referință la **înregistrări** cu structură specificată similar ca în 3.1.a:

- -

```
TYPE TypeElement = RECORD
                    key: integer;                         [3.1.a]
                    {other components}
          END;
```

- -

```
typedef struct {
          int key;
          ... other components;                           /*3.1.a*/
} type_element;
```

- Câmpul *key* care poate nu va fi relevant din punct de vedere al informaților, întrucât informațiile esentiale sunt conținute de celelalte câmpuri ale înregistrării.
- Dar din punct de vedere al sortării, câmup *key* este cel mai important deoarece coonsideăm următoarea **definiție** a sortării:
    - Pentru n elemente

      $a_1, a_2, \ldots, a_n$

    - Sortarea constă în permutarea acestor elemente:

      $a_{k1}, a_{k2}, \ldots, a_{kn}$

    - Astfel încât secvența cheilor să fie **crescătoare**

      $a_{k1}.key \le a_{k2}.key \le \ldots \le a_{kn}.key$

- Câmpul de date *key* sepresupune a fi un număr întreg pentru o înțelegere mai simplă, dar real poate fi orice tip scalar.
- O **metodă de sortare** este **stabilă** dacă ordinea relativă a obiectelor cu aceleași chei rămâne neschimbaată în procesul de sortare.
    - **Stabilitatea sortării** este de obicei de dorit, dacă obiectele sunt deja sortate conform unor chei secundare (i.e. proprietăți care nu reies din cheia primară).
- Dependența alegerii unui algoritm conform structurii de date pentru procesat este un fenomen profund în cazul sortării.
- Din acest motiv,, **metodele de sortare** sunt general clasificate ca:
    - (1) sortarea vectorilor sau **sortare internă**. Obiectele de sortat sunt stocate în regiștri numiți **vectori**.
    - (2) sortarea (secvențială) a fișierelor sau **sortare externă**. Obiectele de sortat sunt stocate în **fișiere** care sunt potrivite pentru registrele mai lente dar mai spațioase (”externe”) bazate pe dispozitive cu părți în mișcare (diskuri, casete).

3.2 Sortarea vectorilor

- Vectorii sunt stocați în **memoria primară** a sistemelor computerizate, de aceea **stocarea vectorilor** se numește **stocare internă**.
- **Cerința predominantă** pentru orice metodă de sortare este **utilizarea economică** a spațiului de stocare.
    - Acest lucru implică ca permutarea obiectelor trebuie să fie ”in situ” – folosirea doar a spațiului alocat vectorului
    - Metodele care transportă obiectele dintr-un vector a într-un vector b sunt de interes minor.
- Având astfel restricționată alegerea de metode din toate soluțiile posibile în funcție de **ecenomia spațiului**, continuăm cu o primă clasificare a algoritmilor de sortare în funcție de **eficiență**, i.e. timpul de execuție.
- Evaluarea cantitativă a eficienței unui algoritm de sortare poate fi exprimat prim **indicatori specifici**:
    - (1) numărul **C** de **comparații ale cheii** pentru sortarea vectorului

- (2) numărul **M** de **mutări** ale elementelor.
  - Ambii indicatori au legătură cu numărul n de elemente pentru sortare.
- Întâi vom discuta câteva metode simple și evidente numite **metode de sortare liniară**, pentru care valorile C și M sunt de ordinul n^2, care însceamnă $O(n^2)$.
- Există **algoritmi avansați de sortare**, de complexitate ridicată, pentru care valorile C și M sunt de ordinul $n * \log_2 n$ ($O(n * \log_2 n)$).
  - Rația $n^2/ (n * \log_2 n)$, care ilustrează eficiența dobândită a acestui algoritm ca aproximativ 10 pentru n=64, respectiv 100 pentru n = 1000.
- În ciuda acestei situații, există motive bune pentru prezentarea **metodelor de căutare liniare** înainte de prezentarea algoritmilor mai rapizi.
  - (1) metodele liniare sun particular potrivite pentru elucidarea caracteristicilor **majorității prinicipilor de sortare**.
  - (2) implementarea lor este scurtă și ușor de înțeles.
  - (3) chiar dacă metode sofisticate necesită mai puține operațiuni, acestea sunt de obicei mai complexe în detaliile lor.
    - În consecință, metodele liniare sunt mai rapide pentru n de dimensiuni ici, însă nu ar trebui folosite pentru n mare.
  - (4) reprezintă punctul de plecare pentru metode avansate de sortare.
- Metodele de sortare care sortează pe loc (in situ) obiecte pot fi clasificare în următoarele categorii principale conform metodei de execuție:
  - (1) sortare prin **inserare**
  - (2) sortare prin **selecție**
  - (3) sortare prin **interschimbare**
- În prezentarea acestora de mai sus vom folosi elementul *TypeElement* descris în 3.1.a precum în următoarele structuri 3.2.a.

- -

```
TYPE TypeIndex = 0..n;
        TypeArray = ARRAY [TypeIndex] OF TypeElement;
VAR a: TypeArray; temp: TypeElement;                                [3.2.a]
```

- -

```
#define N ...
typedef struct {
        int key;                                                    /*3.2.a*/
        ... other components;
} type_element;
type_element a[N];
```

- -

3.2.1 Sortarea prin inserare directă

- Această metodă este folosită în jocurile de cărți.
    - Obiectele (cărțile) sunt conceptual divizate într-o **secvență destinatară** a_1 ... a_{i-1} și **o secvență sursă** a_1 ... a_n .
    - În fiecare pas, începând cu i = 2, elementul i al vectorului (care este primul element al vectorului **sursă**), este ridicat și transferat în **secvența sursă** prin **inserare în** locul potrivit.
    - I este incrementat și ciclul se repetă.
- Întâi primele două elemente sunt sortate, apoi primele trei, etc.
- Trebuie să observăm că în pasul i, primii i-1 obiecte sunt deja sortate, deci procesul de sortare constă doar în inserarea obiectului a[i] în locul său potrivit într-o secvență ordonată. 3.2.1.a

- -

```
{Sortarea prin inserare directă}
FOR i:= 2 TO n DO
      BEGIN                                                  [3.2.1.a]
           temp:=a[i];
           *insert x at the appropriate place in a[1]...a[i]}
END;{FOR}
```

- -

- Pentru înlocuirea lui *a[i], locul potrivit este găsit,* ***secvența destinație*** *deja sortată a[1]...a[i-1] este scanată de la* ***dreapta*** *la* ***stânga*** *comparînd in a[i] cu fiecare element scanat.*
    - În același timp, în timpul scanării, fiecare element testat este depanat către dreapta cu o poziție, pînă la îndeplinirea condiției de oprire.
    - Cu această acțiune, un loc pentru elementul de inserat este făcut în vector.
    - Procesul de scanare este oprit când *in a[j]* având o cheie mai mică sau egală cu in a[i] este găsită.
    - Dacă astfel *in a[j]* nu există, procesul de scanare este oprit la *in a[i]*, prima poziție.
- În cazul tipic al **buclei cu două codniții**, rețineți **metoda santinelă**.

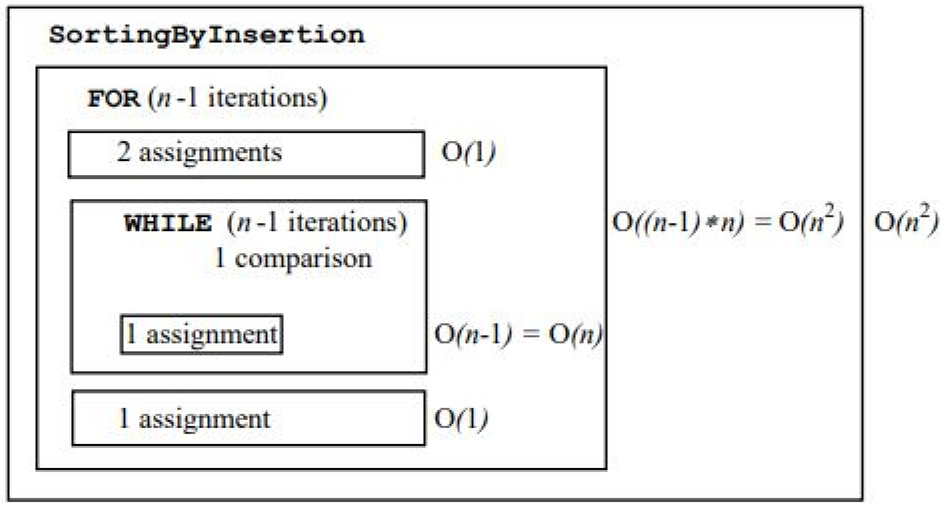
- Pentru asta, **elementul auxiliar** *in a[0]* este adăugat în vector și este onițializat cu valoarea in *a[i].*
- Drept rezultat, condiția - chia lui *in a[j]* să fie mai mic ori egal cu fiecare cheie a lui *in a[i]* - este îndeplinită pentru cel puțin j=0, și nu este necesară verificarea valorii lui j la index j>=0.
- **Inserarea efectivă** este realizată la locația *in a[j+1].*

* Algoritmul corespondent este prezentat în 3.2.1.b și schema temporală este figura 3.2.1.a

- -

```
{Sortare prin inserție directă – varianta Pascal }

PROCEDURE SortingByInsertion;

VAR i,j: TypeIndex; temp: TypeElement;

BEGIN

    FOR i:= 2 TO n DO

        BEGIN

            temp:= a[i]; a[0]:= temp; j:= i-1;

            WHILE a[j].key>temp.key DO

                BEGIN

                    a[j+1]:= a[j]; j:= j-1                    [3.2.1.b]

                END; {WHILE}

             a[j+1]:= temp

    END {FOR}

END; {SortingByInsertion
```

- -



3.2.1.a Schema temporală a algoritmului de sortare prin inserție directă.

- Algoritmul de sortare conține **o buclă externă** condusă de i, care execută n-1 iterații
- În pasul i al buclei **for**:
    - (1) **numărul minim** de iterații în ciclul intern este 0
    - (2) **numărul maxim** este i-1

3.2.1.1 Sortarea prin inserare directă – analiza performanței

- În pasul i al ciclului FOR, numărul C_i al comparațiilor cheii executate în ciclui WHILE depinde de **ordinea inițială** a cheilor.
    - Cel puțin 1 (date ordonate)
    - Cel puțin i-1 (date ordonate invers)
    - În medie i/2, presupunând că toate permutațiile ale celor n chei sunt posibile în mod egal.

- Doarece avem n-1 iterații în ciclui FOR pentru i = 2, 3, ..., n, indicatorul C poate obține valorile prezentate în 3.2.1.c

$$C_{min} = \sum_{i=2}^{n} 1 = n-1$$

$$C_{max} = \sum_{i=2}^{n} (i-1) = \sum_{i=1}^{n-1} i = \frac{(n-1)\cdot n}{2} \qquad [3.2.1.c]$$

$$C_{avg} = \frac{C_{min} + C_{max}}{2} = \frac{n^2+n-2}{4}$$

- Numărul de **măscări M_i** în ciclul FOR este $C_i + 3$.
    - **Explicație**: la numărul C_i, mișcările executate în bucla WHILE sunt de tipul *a[j+1] = a[j]*, și 3 mișcări suplimentare sunt adăugate (temp = *a[i], a[0] = temp, și a[i+1] = temp.*
    - Chiar și pentru numărul minim de comparații care este 0, cele 3 atribuiri menționate rămân valabile.
- Ca și rezultat, indicatorul M poate lua valorile din 3.2.1.d

$$M_{min} = 3\cdot(n-1)$$

$$M_{max} = \sum_{i=2}^{n} (C_i + 3) = \sum_{i=2}^{n} (i+2) = \sum_{i=1}^{n+2} i - (1+2+3) = \frac{(n+2)\cdot(n+3)}{2} - 6 = \frac{n^2+5\cdot n-6}{2} \qquad [3.2.1.d]$$

$$M_{avg} = \frac{M_{min} + M_{max}}{2} = \frac{n^2+11\cdot n-12}{4}$$

- Putem observa că valorile C și M sunt de ordineul n^2 ($O(n^2)$).
- **Numerele minime** au loc dacă obiectele sunt **inițial ordonate**, sau **cel mai rău caz** are loc dacă obiectele sunt inițial **invers ordonate**.
- **Sortarea prin insterție directă este o sortare stabilă**.
- **În 3**.2.1.e este prezentată o modificare ușoară (în limbajul C) a acestei metode de sortare.

- -

```
// Sortarea prin inserție directă –  varianta  C
StraightInsertion(int a[],int n){//sentinel on position a[n]
        for(int i=n-2;i>=0;i--) {
                a[n]=a[i];
                int j=i+1;
                while(a[j]<a[n]) {
                        a[j-1]=a[j]; j++;                    /*3.2.1.e*/
                }
                a[j-1]=a[n];
        }
}
```

- -

- Referitor la 3.2.1.c, facem următoarele observații:
  - Implementarea este **în oglindă** comparând cu varianta pascal.
  - Vectorul conține n elemente *a[0]... a[n-1];*
  - **Secvența sursă** este *a[0]... a[i];*
  - **Secvența destinatar** (cea ordonată) este *a[i+1]... a[n-1];*
  - **Santinela** este poziția n din vectorul a.
  - În procesul găsirii locului pentru inserare, în pasul curent, secvența destinatar este scanată folosind indexul j de la **stânga** la **dreapta**, respectiv începând cu poziție i+1 până la găsirea locului de inserare sau poziția n este găsită.
  - Obiectele întâlnite care ai chei mai mici sau egale cu cheia de inserare sunt **depanate la stânga** cu o poziție, până când condiția este împlinită.

3.2.1.2 Sortarea prin inserare binară

- Algoritmul **inserării directe** poate fi ușor îmbunătățit cu ordonarea **secvenței destinatar** *a[0]... a[i-1];*
- În acest caz, o metodă mai rapidă de a determina **punctul de inserție** este folosirea **căutării binare**.

- Astfel despărțim în două părți egale intervalul de căutare, până ce punctul de isnerare este găsit.

- Algoritmul modificat este **inserarea binară** 3.2.1.f.

- -

```
{Sortarea prin inserarea binară – Varianta Pascal}
PROCEDURE SortingByBinaryInsertion;
VAR i,j,left,right,m: TypeIndex;
    temp: TypeElement;
    a: TypeArray;
BEGIN
    FOR i:= 2 TO n DO
        BEGIN
            temp:= a[i]; left:= 1; right:= i-1;
            WHILE left<=right DO
                BEGIN [3.2.1.f]
                    m:= (left+right)DIV 2;
                    IF a[m].key>temp.key THEN
                        right:= m-1
                     ELSE
                        left:= m+1
                     END;{WHILE}
                FOR j:= i-1 DOWNTO left DO a[j+1]:= a[j];
                a[left]:= temp
     END {FOR}
END; {SortingByBinaryInsertion}
```

- -

```
/* Sortarea prin inserarea binară – Varianta C*/
void sorting_by_binary_insertion()
{
    type_index i,j,left,right,m;
    type_element temp;
    type_array a;
    for(i=2; i<=n; i++)
    {
        temp= a[i]; left= 1; right= i-1;
        while (left<=right)
        {                                                    /*[3.2.1.f]*/
            m= (left+right)/ 2;
            if (a[m].key>temp.key)
                right= m-1;
            else
                left= m+1;
        } /*while*/
        for( j= i-1; j >= left; j --) a[j+1]= a[j];
            a[left]= temp;
    } /*for*/
} /* sorting_by_binary_insertion */
```

- -

3.2.1.3 Sortarea prin inserare binară – analiza performanței

- În cazul sortării prin inserăre binară, poziția de innserare este găsită dacă:
  a[j].key ≤ x.key ≤ a[j+1].key
  ce înseamnă că intervalul de căutare are dimensiunea 1.

- Dacă intervalul de căutare are lungimea i, sunt necesari $\log_2(i)$ pași pentru determinarea poziției de inserare.
- Datorită lungimii intervalului de căutare în fiecare pas este i, avem n pași, deci totalul numărului de comparații C executați în ciclul FOR exterior este prezentat în 3.2.1.g

$$C = \sum_{i=1}^{n} \lceil \log_2 i \rceil$$

- Suma poate fi aproximată intergând precum în 3.2.1.h

$$C = \int_{1}^{n} \log_2 x \cdot dx = x \cdot (\log_2 x - c) \Big|_1^n = n \cdot (\log_2 n - c) + c$$

$$c = \log_2 e = 1 / \ln 2 = 1.44269$$

- Numărul comparațiilor este esențial **independent** de ordinea inițială a obiectelor.
    - Acest lucru nu este oișnuit pentru un algoritm de sortare.
- Din nefericire, îmbunătățirea obținută folosind căutarea binară se aplică numai **numărului de comparații** dar nu și **numărului de mișcări necesare**.
- Întrucât **mișcarea elementelor**i.e. cheile și informațiile asociate, este în general **mai costisitoare din punct de vedere al timpului** necesar decât compararea a două chei, îmbunătățirea nu este drastică:
    - M este de ordinul n^2.
    - Sortarând vectorul deja sortat durează mai mult timp decât inserarea directă cu căutare directă.
- În concluzie, sortând prin inserție nu este o metodă potrivită de sortare folosind un sistem computerizat, doarece inserarea unui element presupune **depanarea cu o poziție** a numărului de elemente, care nu este nici recomandată și nici eficientă.
    - Se așteaptă **rezultate îmbunătățite** de la o metodă în care mișcarea elementelor se face individual fiecătui element și **pe distanțe mari**.
- Acestă idee duce la **sortarea prin selecție**.

3.2.2 Sortarea prin selecție

- Sortarea prin **selecție directă** este bazată pe ideea selectării elementului cu cheia minimă și interschimbării poziției acestui element cu elementul din prima poziție.
    - Acest procedeu este repetat pentru toate cele n-1 elemente rămase.
- Ne reamintim că **sortarea prin selecție directă** presupune că toate elementele a sursei sursă în care căutarea are loc.
- Contrar, **sortărea prin selecție directă** presupune ca toate elementele din secvența sursă în care se face căutarea elementului ce cheie minimă și plasarea lui ca următorul element în secvența destinație.

- -

```
{Sortarea prin selecție directă}
FOR i:= 1 TO n-1 DO                                          [3.2.2.a]
     BEGIN
          *find the smallest item of the a[i]...a[n] and assign
           variable min with its index;
          *interchange a[i] with a[min]
 END;
```

- -

- Schema temporată a algoritmului prezentat în 3.2.2.b este prezentată în figura 3.2.2.a.

- -

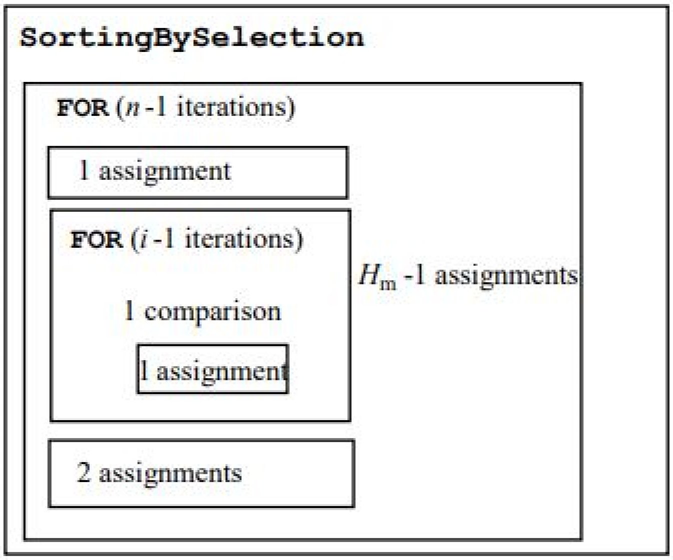
```
{Sorting by straight selection – Pascal Variant}
PROCEDURE SortingBySelection;
VAR i,j,min: TypeIndex; temp: TypeElement;
     a: TypeArray;
BEGIN
     FOR i:= 1 TO n-1 DO
          BEGIN
               min:= i; temp:= a[i];
               FOR j:= i+1 TO n DO
                    IF a[j].key<temp.key THEN                 [3.2.2.b]
                         BEGIN
                              min:= j; temp:= a[j]
                        END;{FOR}
```

```
                    a[min]:= a[i]; a[i]:= temp
          END {FOR}
END; {SortingBySelection}
```

- -

```
/* Sorting by straight selection – C Variant */
void sorting_by_selection()
{
     typeindex i,j,min; typeelement temp;
     typearray a;
      for(i=1; i<=n-1; i++)
      {
           min= i; temp= a[i];
          for(j=i+1; j<=n; j++)
           if(a[j].key<temp.key)                              /*[3.2.2.b]*/
          {
                min= j; temp= a[j];
           } /*for*/
      a[min]= a[i]; a[i]= temp;
      } /*for*/
} /*sorting_by_selection*
```

- -



3.2.2.a Schema temporală a algoritmului de sortare prin selecție

3.2.2.1 Sortarea prin selecție – analiza performanței

- Evident, numărul C al comparațiilor de chei este independent de ordinea inițială a cheilor. Este **fixat** determiant de executarea integrală a celor două cicluri FOR 3.2.2.c
  - În acest caz, metoda se poate spune că se comportă mai puțin natural decât inserarea directă

$$C = \sum_{i=1}^{n-1}(i-1) = \sum_{i=1}^{n-2} i = \frac{n^2 - 3\cdot n + 2}{2} \qquad [3.2.2.c]$$

- Numărul M de mutări este cel puțin 3 pentru fiecare valoare a lui i,
  temp:= a[i],a[min]:= a[i],a[i]:= temp, deci rezultă:

$$M_{min} = 3\cdot(n-1) \qquad [3.2.2.d]$$

  - Acest minim devine eficient în cazul **sortării inițiale** a cheilor.
- Dacă aceste chei sunt inițial invers sortate, M_{max} poate fid eterminat folosind **formula empirică** 3.2.2.e

$$M_{max} = \left\lfloor \frac{n^2}{4} \right\rfloor^{(1)} + 3\cdot(n-1) \qquad [3.2.2.e]$$

- Valoarea indicatorului M_{avg} **NU** este media dintre M_{min} și M_{max} .
- Pentru determinarea M_{avg} facem următoarele deliberări:

- Algoritmul scanează vectorul care conține m elemente, comparând fiecare element cu valoare minimă detectat până într-un moment și, dacă este mai mic decât minimul, efectuează atribuirea.
- Probabilitatea ca elementul secund să fie mai mic sau egal este 1/2. Acesata este și probabilitatea pentru a atribui un nou minim.
- Șansa ca al treilea element să fie mai mica decât primii doi este 1/3.
- Șansa ca al patrulea element să fie cel mai mic dintre primii trei este 1/4 și așa mai departe.
- Prin urmare, **numărul total de mișcări estimate** pentru un vector care conține m elemente este H_{m-1} unde H_m este al m-lea **număr armonic** . 3.2.2.f

$$H_m = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{m} \qquad [3.2.2.f]$$

- Această valoare reprezintă **numărul total de mutări estimate**, ceea ce înseamnă numărul de atribuiri a variabilei temp, deoarece procesul de sortarea unei secvențe de m elemente din ciclul FOR interori, temp este atribuit oricând un obiect este găsit mai mic decât toate obiectele precedente.
- Trebuie să adăugăm acestei valori constanta 3 care reprezintă atribuirile *temp:=a[i],a[min]:=a[i] și a[i]:=temp.*
- Drept rezultat, **valoarea medie a numărului total de mutări estimate** la scanarea secvenței care conține m elemente este H_m +2.
- Este demonstrat că **seria este divergentă**, dar putem calcula suma parțială folosind fromula lui Euler. 3.2.2.g:

$$H_m \approx \ln m + \gamma + \frac{1}{2 \cdot m} - \frac{1}{12 \cdot m^2} + \frac{1}{120 \cdot m^4} \qquad [3.2.2.g]$$

where $\gamma = 0.5772156649...$ is **Euler's** constant [Kn76].

- Pentru un m destul de mare, valoarea lui H_m poate fi aproximată prin expresia 3.2.2.h:

$$H_m \approx \ln m + \gamma \qquad [3.2.2.h]$$

- Tot ce am discutat până acum este valabil pentru o singură scanare a secvenței de m chei.
- Deoarece în procesul de sortare sunt scanate consecutiv n secvențe care au lungimi m = n, n-1, n-2, ..., 1 fiecare necesitând H_m +2 mutări, numărul mediu demutări este H_{avg} 3.2.2.i.

$$M_{avg} \approx \sum_{m=1}^{n} (H_m + 2) \approx \sum_{m=1}^{n} (\ln m + \gamma + 2) = n \cdot (\gamma + 2) + \sum_{m=1}^{n} \ln m \qquad [3.2.2.i]$$

- Suma poate fi aproximată foosind calculul:

$$\int_1^n \ln x \cdot dx = x \cdot (\ln x - 1)\Big|_1^n = n \cdot \ln(n) - n + 1 \qquad [3.2.2.j]$$

- Iar rezultatul final este:

$$M_{avg} \approx n \cdot (\ln m + \gamma + 1) + 1 = O(n \cdot \ln n) \qquad [3.2.2.k]$$

- Putem conclude că în general **algoritmul selecției directe** este preferat **inserției directe**.
    - Chiar dacă, în cazurile în care cheile sunt inițial sortate ori aproape sortate, inserarea directă este mai rapidă
- Optimizarea performanței sortării poate fi obținută prin reducerea numărului de mutări.
- Sedgewik propune o variantă care în loc să stocheze minimul curent în variabila temp, va memora indexul. Mutarea efectivă se va face numai pentru ultimul minim determinat, după finalizarea buclei FOR interioare.

- -

```
{Sortarea prin selecție optimizată - Varianta Pascal}
PROCEDURE OptimizedSelection;
VAR i,j,min: TypeIndex; temp: TypeElement;
         a: TypeArray;
BEGIN
          FOR i:= 1 TO n-1 DO [3.2.2.l]
                   BEGIN
                            min:= i;
                           FOR j:= i+1 TO n DO
                                    IF a[j].key<temp.key THEN min:= j;
                           temp:= a[min]; a[min]:= a[i]; a[i]:= temp
          END {FOR}
END; {OptimizedSelection}
```

- -

```
/* Sortarea prin selecție optimizată - Varianta C*/
void optimized_selection()
{
    type_index i,j,min;
    type_element temp;
    type_array a;
    for(i= 1; i <= n-1; i ++) /*[3.2.2.l]*/
    {
        min= i;
        for(j= i+1; j <= n; j ++)
            if(a[j].key<temp.key) min= j;
        temp= a[min]; a[min]= a[i]; a[i]= temp;
    } /*for*/
} /*optimized_selection*/
```

- -

- Din nefericire, măsurătorile experimentale efectuate asupra acestui algoritm nu evidențiază nicio îmbunătățire asupra performanței chiar și pentru dimensiuni mari de vectori pentru sortat.
    - Explicație: nu există nicio diferență între atribuirea normalp și o atribuire care presupune accesul în variabila indexată.

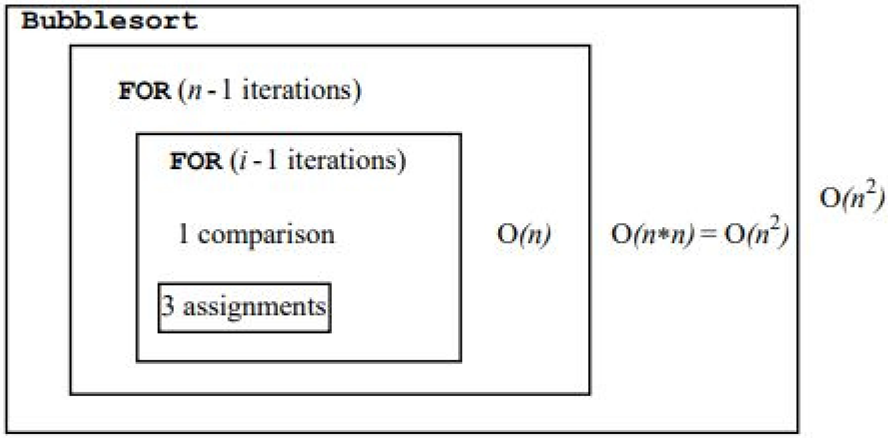
3.2.3 Sortarea prin interschimbare directă. Bubblesort și shakersort.

- Sortările sunt bazate pe metode: de inserție, de selecție, de interschimbare. Precedentele două sortări sunt sortări prin interschimbare.
- Algorimtul interschimbprii directe se bazează pe compararea și interschimbarea perechilor de elemente adiacente până când toate elementele sun sortate.
- În algoritmii precedenți, se fac treceri repetate peste vectori depanând toate elementele la stânga.

- Dacă vizualizăm vectorul vertical, iar elementele drept bule de aer sub apă, fiecare trecere peste vector va rezulta ridicarea bulei de aer la locul său potrivit, de unde și unmele metodei – bubblesort. 3.2.3.a

```
{Sorting by exchange: Bubblesort - Variant 1}

PROCEDURE Bubblesort;

VAR i,j: TypeIndex; temp: TypeElement;

BEGIN
  FOR i:= 2 TO n DO
    BEGIN                                        [3.2.3.a]
      FOR j:= n DOWNTO i DO
        IF a[j-1].key>a[j].key THEN
          BEGIN
            temp:= a[j-1]; a[j-1]:= a[j]; a[j]:= temp
          END {IF}
    END {FOR}
END; {Bubblesort}
```

```
/* Sorting by exchange: Bubblesort - Variant 1*/

void bubblesort()
{
    type_index i,j; type_element temp;

  for(i=2; i<=n; i++)
    {                                       /*[3.2.3.a]*/
      for(j= n; j>=i; j--)
        if (a[j-1].key>a[j].key)
          {
            temp= a[j-1]; a[j-1]= a[j]; a[j]= temp;
          }
    }
} /*bubblesort*/
```



3.2.a Schema temporală a algoritmului de sortare prin interschimbare.

- Elemente importante de notificat:
  - 1. În multe cazuri, procesul de sortare se încheie înaintea finalizării buclei externe FOR. Algoritmul poate îmbunătățit prin memorarea cazului în care o interschimbare a avut loc în acea trecere.

```
{ Sorting by exchange: Bubblesort - Variant 2}

PROCEDURE Bubblesort1;
VAR i: TypeIndex; modified: boolean;
    temp: TypeElement;
BEGIN
  REPEAT
    modified:= false;
    FOR i:= 1 TO n-1 DO
      IF a[i].key>a[i+1].key THEN           [3.2.3.b]
        BEGIN
          temp:= a[i]; a[i]:= a[i+1]; a[i+1]:= temp;
          modified:= true
        END
  UNTIL NOT modified
END; {Bubblesort1}
------------------------------------------------------------
/* Sorting by exchange: Bubblesort - Variant 2*/
typedef int boolean;
#define true (1)
#define false (0)

void bubblesort1()
{
    type_index i; boolean modified;
    type_element temp;

  do {
    modified= false;
    for(i=1; i<=n-1; i++)
      if (a[i].key>a[i+1].key)        /*[3.2.3.b]*/
        {
          temp= a[i]; a[i]= a[i+1]; a[i+1]= temp;
          modified= true;
        }
  } while (!(! modified));
} /*bubblesort1*/
```

- 2. Această îmbunătățire poate fi mai nedaprte îmbunătățită prin memorarea indexului k a ultimei interschimbări.
  - Este evident că toate perchile de elemente adiacente după inexul k sunt ordonate.
- 3. Asimetria – un singur element dezordonat și plasat în capătul opus al vectorului sortat se va așeza la locul potrivit într-o singură trecere.
  12 18 22 34 65 67 83 <u>04</u>
  se va sorta folosing bublesort varianta 2 într=o singură trecere.

În schimb,

83 12 18 22 34 65 67 04

necesită 7 pași pentru sortare. Această asimetrie sugerează a treia îmbunătățire: alternarea direcției de sortare între treceri.

```
{Sorting by exchange - Variant 3}

PROCEDURE Shakersort;
VAR j,last,up,down: TypeIndex;
    temp: TypeElement;
BEGIN
  up:= 2; down:= n; last:= n;
  REPEAT
    FOR j:= down DOWNTO up DO                        [3.2.3.c]
      IF a[j-1].key>a[j].key THEN
        BEGIN
          temp:= a[j-1]; a[j-1]:= a[j]; a[j]:= temp;
          last:= j
        END;{FOR}
    up:= last+1;
    FOR j:=up TO down DO
      IF a[j-1].key>a[j].key THEN
        BEGIN
          temp:=a[j-1]; a[j-1]:=a[j]; a[j]:=temp;
          last:=j
        END;{FOR}
    down:=last-1
  UNTIL (up>down) {REPEAT}
END; {Shakersort}
```

```
/* Sorting by exchange - Variant 3*/

void shakersort()
{
    type_index j,last,up,down;
    type_element temp;

  up= 2; down= n; last= n;
  do {
    for(j=down; j>= up; j--)              /*[3.2.3.c]*/
      if (a[j-1].key>a[j].key)
        {
          temp= a[j-1]; a[j-1]= a[j]; a[j]= temp;
          last= j;
```

```
      } /*for*/
  up= last+1;
  for(j=up; j<= down; j++)
    if (a[j-1].key>a[j].key)
      {
        temp=a[j-1]; a[j-1]=a[j]; a[j]=temp;
        last=j;
      } /*for*/
  down=last-1;
 } while (!(up>down));
}    /*shakersort*/
/*-------------------------------------------------------*/
```

3.2.3 Sortarea prin interschimbare directă. Bubblesort și shakersort. – analiza de performanță

- Numărul de comparații pentru bublesort este constant:

$$C = \sum_{i=1}^{n-1}(i-1) = \frac{n^2 - 3\cdot n + 2}{2} \qquad [3.2.3.d]$$

- Numărul minim, maxim, și mediu de mutări este:

$$M_{min} = 0$$

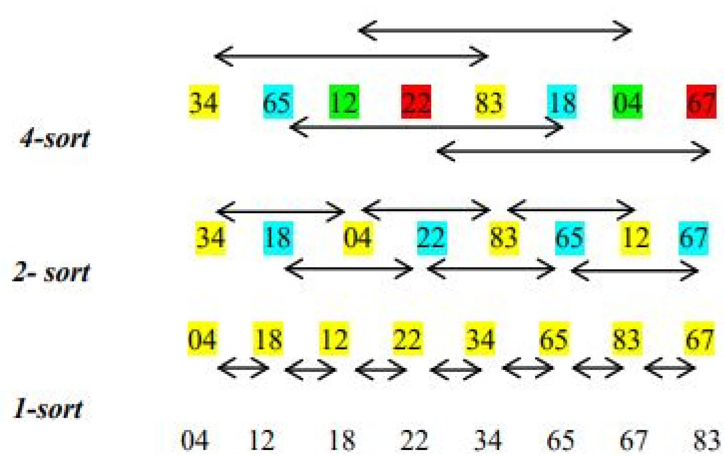
$$M_{max} = 3 \cdot C = \frac{3}{2} \cdot (n^2 + 3 \cdot n + 2) \qquad [3.2.3.e]$$

$$M_{avg} = \frac{3}{4}(n^2 + 3 \cdot n + 2)$$

- Analiza performanței **shakersort** duce la $C_{min} = n-1$.
  - Numărul mediu de treceri este proporțional cu $n - k_i * sqrt(n)$ și un număr mediu de comparații $C_{med} = 1/2\ (n^2 - n\ (k_2 + \ln n))$.
- Toate îmbunătățirile prezentate **nu** îmbunătățesc numărul de interschimbări. Numai reduc numărul de verificări duble.
- Analza comparativă a performanței algoritmilor prezentați conclude că:
  - 1 – sortarea prin interschimbare este inferioară ca performanță sortării prin inserție
  - 2 – shakersort este avantajoasă atunci când cunoaștem că elementele sunt aproape ordonate
- În medie, numărul de locații pe care peste care fiecare element călătorește este n/3.
- Toate metodele directe de sortare mută fiecare element cu o poziție la fiecare iterație.
  - Astfel, sunt necesari n^2 astfel de pași.
- O îmbunătățire este mărirea ”pașilor” făcuți de fiecare element

3.2.4 Sortarea prin inserare cu increment scăzător. Shellsort.

- O rafinare a sortării prin inserare directă a fost propusă de D.L. Shell în 1959.



3.2.4 Sortarea prin inserare cu increment scăzător