3. Sortări

3.1. Conceptul de sortare

- **Scopul** fundamental al acestui capitol este:
  - De a furniza un **set extins de exemple** referitoare la utilizarea structurilor de date introduse în capitolul 1;
  - De a sublinia **influența** profundă pe care adoptarea unei anumite **structuri** o are asupra **algoritmului** care o utilizează şi asupra **tehnicilor de programare** care implementează algoritmul respectiv.
- Sortarea este domeniul ideal al **studiului:**
  - **(1) Construcției** algoritmilor
  - (2) **Performanțelor** algoritmilor
  - (3) **Avantajelor** şi **dezavantajelor** unor algoritmi față de alții în accepțiunea unei aplicații concrete
  - (4) **Tehnicilor de programare** aferente diferiților algoritmi
- Prin **sortare** se înțelege în general **ordonarea** unei mulțimi de elemente, cu scopul de a facilita **căutarea** ulterioară a unui element dat.
  - Sortarea este o activitate **fundamentală** cu caracter **universal**.
  - În general în orice situație în care trebuiesc căutate şi regăsite obiecte, sortarea este prezentă.
- În continuare se presupune că sortarea se referă la anumite elemente care au o structură articol definită după cum urmează [3.1.a]:

```
------------------------------------------------------------
      TYPE TipElement = RECORD
                           cheie: integer;         [3.1.a]
                           {Alte câmpuri definite}
                          END;
------------------------------------------------------------
typedef struct tipelement {
            int cheie;          /*[3.1.a]*/
            /*Alte câmpuri definite*/
} tipelement;
------------------------------------------------------------
```

- Câmpul `cheie` precizat, poate fi neesențial din punctul de vedere al informației înregistrate în articol, partea esențială a informației fiind conținută în celelalte câmpuri.
- Din punctul de vedere al sortării însă, `cheie` este cel mai important câmp întrucât este valabilă următoarea **definiție** a sortării.
  - Fiind dat un şir de elemente aparținând tipul mai sus definit

    $a_1,\ a_2, \ldots, a_n$

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

$a_{k1}, a_{k2}, ..., a_{kn}$

- Astfel încât şirul cheilor să devină **monoton crescător**, cu alte cuvinte să avem

  $a_{k1}.cheie \le a_{k2}.cheie \le ... \le a_{kn}.cheie$

- Tipul câmpului `cheie` se presupune a fi întreg pentru o înţelegere mai facilă, în realitate el poate fi însă orice tip scalar.
- O metodă de sortare se spune că este **stabilă** dacă după sortare, ordinea relativă a elementelor cu chei egale coincide cu cea iniţială
  - Această stabilitate este esenţială în special în cazul în care se execută sortarea după mai multe chei.
- În cazul sortării, 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 elementele de sortat:
  - (1) Sunt înregistrate ca şi tablouri în **memoria centrală** a sistemului de calcul ceea ce conduce la **sortarea tablourilor** numită **sortare internă**
  - (2) Sunt înregistrate într-o **memorie externă**: ceea ce conduce la **sortarea fişierelor** (secvenţelor) numită şi **sortare externă.**

3.2. 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.
- Aprecierea **cantitativă** a eficienţei unui algoritm de sortare se realizează prin intermediul unor **indicatori specifici**.
  - (1) Un prim indicator este **numărul_comparaţiilor de chei** notat cu **C**, pe care le execută algoritmul în vederea sortării.
  - (2) Un alt indicator este **numărul de atribuiri 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.
- 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 $n = 1000$.
- Cu toate că ameliorarea este substanţială, metodele de sortare directe prezintă interes din următoarele motive:
  - (1) Sunt foarte potrivite pentru explicitarea principiilor majore ale sortării.

- (2) Procedurile 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.

- Metodele de sortare directe care realizează sortarea "**in situ**" se pot clasifica în trei mai categorii:
  - (1) Sortarea prin **inserţie;**
  - (2) Sortarea prin **selecţie;**
  - (3) Sortarea prin **interschimbare**.
- În prezentarea acestor metode se va lucra cu tipul element definit anterior, precum şi cu următoarele notaţii [3.2.a].

```
------------------------------------------------------------
TYPE TipIndice = 0..n;
     TipTablou = ARRAY [TipIndice] OF TipElement;
VAR  a: TipTablou; temp: TipElement;              [3.2.a]
------------------------------------------------------------
#define N  ...
typedef struct tipelement {
            int cheie;
            /*Alte câmpuri definite*/
} tipelement;
typedef tipelement tiptablou[N];
tiptablou a; tipelement temp;                     /*[3.2.a]*/
------------------------------------------------------------
```

3.2.1. Sortarea prin inserţie

- 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 = 2`, elementul `i` al tabloului (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.
- Astfel, la început se sortează primele două elemente, apoi primele trei elemente ş.a.m.d.
- Se face precizarea că în pasul `i`, primele `i-1` elemente sunt deja sortate, astfel încât sortarea constă numai în a insera elementul `a[i]` la locul potrivit într-o secvenţă deja sortată.
- În termeni formali, acest algoritm este precizat în secvenţa [3.2.1.a].
- Se precizează faptul că elementele tabloului se consideră poziţionate de la `a[1]` până la `a[n]`. Poziţia `a[0]` va fi utilizată în alte scopuri.

```
------------------------------------------------------------
{Tehnica sortării prin inserţie}
 FOR i:= 2 TO n DO
   BEGIN                                        [3.2.1.a]
     temp:=a[i];
     *inserează x la locul potrivit în a[1]...a[i]}
   END;{FOR}
------------------------------------------------------------
```

* **Selectarea** locului în care trebuie inserat `a[i]` se face parcurgând secvența destinație deja sortată `a[1],...,a[i-1]` de la dreapta la stânga
    * Oprirea 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[1]` adică pe prima poziție.
* **Simultan** cu parcurgerea, se realizează şi **deplasarea spre dreapta** cu o poziție a fiecărui element testat până în momentul îndeplinirii condiției de oprire.
* Acest caz tipic de repetiție cu două condiții de terminare readuce în atenție **metoda fanionului**.
    * Pentru aplicarea ei se introduce elementul auxiliar `a[0]` care se asignează inițial cu `a[i]`.
    * În felul acesta, cel mai târziu pentru `j = 0`, condiția de a avea cheia lui `a[j]` "mai mică sau egală" cu cheia lui `a[i]` se găseşte îndeplinită.
    * Inserția propriu-zisă se realizează pe poziția `j+1`.
* Algoritmul care implementează sortarea prin inserție apare în [3.2.1.b] iar profilul său temporal în figura 3.2.1.a.
* Se precizează faptul că elementele tabloului se consideră poziționate de la `a[1]` până la `a[n]`. Poziția `a[0]` este utilizată pe post de fanion.

```
----------------------------------------------------------------
/*Sortarea prin insertie*/
tipelement a[n+1];
void sortare_prin_insertie()
{
  int i,j; tipelement temp;

  for( i= 2; i <= n; i ++)
    {
      temp= a[i]; a[0]= temp; j= i-1;
      while (a[j].cheie>temp.cheie)
        {
          a[j+1]= a[j];  j= j-1;              /*[3.2.1.b]*/
        }
      a[j+1]= temp;
     }
}
----------------------------------------------------------------
```

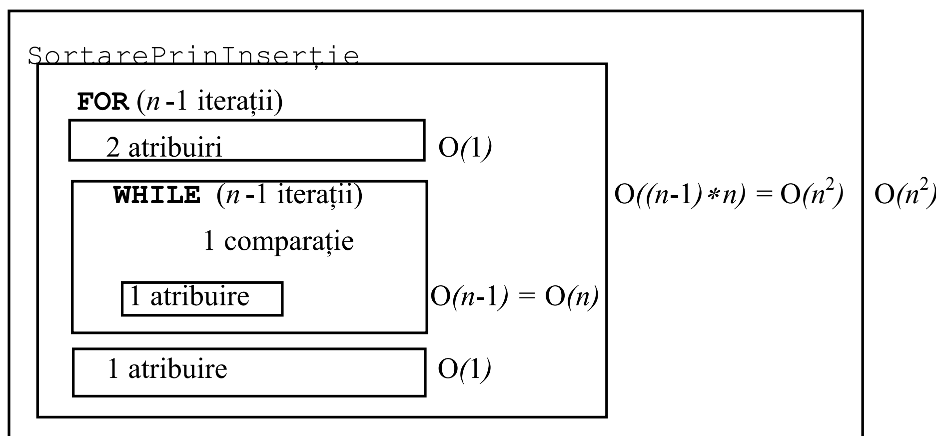


Fig.3.2.1.a. Profilul temporal al algoritmului de sortare prin inserție

- După cum se observă, algoritmul de sortare conține un ciclu exterior după `i` care se reia de `n-1` ori (bucla **`FOR`**).
  - În cadrul fiecărui ciclu exterior se execută un ciclu interior de lungime variabilă după `j`, până la îndeplinirea condiției (bucla **`WHILE`**).
  - În pasul `i` al ciclului exterior **`FOR`**, numărul minim de reluări ale ciclului interior este `0` iar numărul maxim de reluări este `i-1`.

- ## Analiza sortării prin inserție
  - În cadrul celui de-al i-lea ciclu **`FOR`**, numărul C_i al comparațiilor de chei executate în bucla **`WHILE`**, depinde de ordinea inițială a cheilor, fiind:
    - Cel puțin `1` (secvența ordonată),
    - Cel mult `i-1` (secvența ordonată invers)
    - În medie `i/2`, presupunând că toate permutările celor `n` chei sunt egal posibile
  - Întrucât avem `n-1` reluări ale lui **`FOR`** pentru `i:= 2..n`, parametrul `C` are valorile precizate în [3.2.1.c].

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

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

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

- Numărul maxim de **atribuiri de elemente** M_i în cadrul unui ciclu **`FOR`** este $C_i + 3$ şi corespunde numărului maxim de comparații
  - **Explicația**: la numărul C_i de atribuiri executate în cadrul ciclului interior **`WHILE`** de tip `a[j+1]:= a[j]` se mai adaugă 3 atribuiri (`temp:= a[i]`, `a[0]:= temp` şi `a[i+1]:= temp`).
  - Chiar pentru numărul minim de **comparații** de chei (C_i egal cu 1) cele trei atribuiri rămân valabile.
- În consecință, parametrul M ia următoarele valori [3.2.1.d].

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

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

$$= \frac{(n+2)\cdot(n+3)}{2} - 6 = \frac{n^2+5\cdot n-6}{2}$$

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

- Se observă că atât C cât şi M sunt de ordinul lui n^2 ($O(n^2)$).
- Valorile **minime** ale indicatorilor rezultă când `a` este **ordonat**, iar valorile **maxime**, când `a` este **ordonat invers**.
- Sortarea prin inserție este o **sortare stabilă**.
- În secvența [3.2.1.e] se prezintă o altă variantă a acestei metode de sortare.

```
// Sortarea prin inserţie - varianta C

insertie(int a[],int n) { //fanionul pe poziţia 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];
 }
}
```

- Relativ la această secvență se fac următoarele observații
  - Implementarea este varianta "în oglindă" față de varianta anterioară. Tabloul va începe de la poziția `0` şi nu de la poziția `1` ca şi în cazul anterior
  - Tabloul de `n` elemente este `a[0]...a[n-1]`
  - Secvența sursă este `a[0]...a[i]`
  - Secvența destinație (cea ordonată) este `a[i+1]..a[n-1]`
  - Fanionul este poziționat pe poziția `n` a tabloului `a`
  - În procesul de căutare a locului de inserție, în pasul curent, se parcurge cu indicele `j` secvența destinație de la poziția `i+1` până la găsirea locului inserției sau până la `n`
  - Elementele întâlnite care sunt mai mici ca şi cheia de inserat se mută cu o poziție spre stânga până la îndeplinirea condiției

3.2.2. Sortarea prin selecție

- Sortarea prin selecție foloseşte procedeul de a **selecta** elementul cu cheia minimă şi de a schimba între ele poziția acestui element cu cea a primului element.
- Se repetă acest procedeu cu cele n-1 elemente rămase, apoi cu cele n-2, etc., terminând cu ultimele două elemente.
- Metoda este oarecum opusă sortării prin inserție care presupune la fiecare pas un singur element al secvenței sursă şi toate elementele secvenței destinație în care se caută de fapt locul de inserție.
- Selecția în schimb presupune toate elementele secvenței sursă dintre care selectează pe cel cu cheia cea mai mică şi îl depozitează ca element următor al secvenței destinație.

```
------------------------------------------------------------------
{Tehnica sortarii prin selectie}
FOR i:= 0 TO n-2 DO                                [3.2.2.a]
  BEGIN
    *găseste cel mai mic element al lui a[i]...a[n] şi
        asignează pe min cu indicele lui;
    *interschimbă pe a[i] cu a[min]
  END;
------------------------------------------------------------------
```

- În urma procesului de detaliere rezultă algoritmul prezentat în [3.2.2.b] al cărui profil temporal apare în figura 3.2.2.a.

```
------------------------------------------------------------------
/*Sortare prin selecţie*/
void sortare_prin_selectie()
{
    int i,j,min; tipelement temp;
    tipelement a[n];
  for( i= 0; i < n-1; i ++)
    {
      min= i; temp= a[i];
      for( j= i+1; j < n; j ++)
        if (a[j].cheie<temp.cheie)             /*[3.2.2.b]*/
          {
            min= j; temp= a[j];
          }
      a[min]= a[i]; a[i]= temp;
    }
}
------------------------------------------------------------------
```

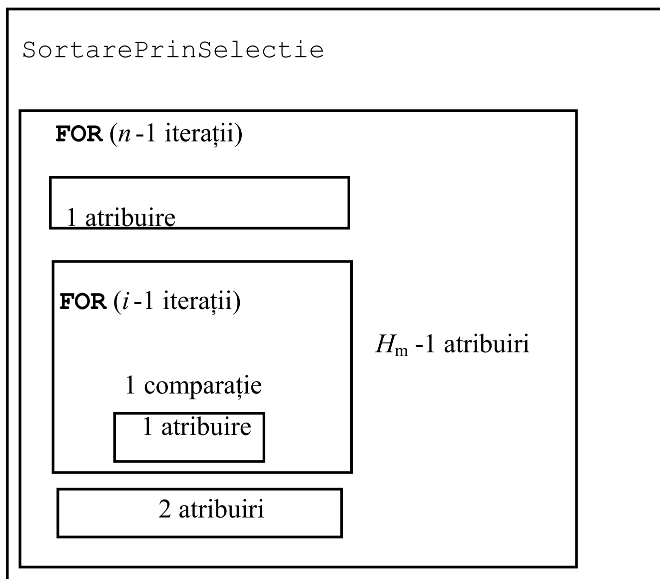


Fig.3.2.2.a. Profilul temporal al sortării prin selecţie

- **Analiza sortării prin selecție**

- Numărul **comparațiilor** de chei C este independent de ordinea inițială a cheilor. El este fix fiind determinat de derularea celor două bucle `FOR` încuibate.

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

- Numărul minim al **atribuirilor** este de cel puțin 3 pentru fiecare valoare a lui `i`, (`temp:=a[i],a[min]:=a[i],a[i]:=temp`), de unde rezultă:

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

  - Acest minim poate fi atins efectiv, dacă inițial cheile sunt deja sortate.
- Pe de altă parte dacă cheile sunt inițial în ordine inversă M_{max} se determină cu ajutorul **formulei empirice** [3.2.2.e] [Wi76].

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

- **Valoarea medie** a lui M **nu** este media aritmetică a lui M_{min} și M_{max}, ea obținându-se printr-un **raționament probabilistic**.
  - Acest raționament probabilistic conduce la rezultatul [Cr00]:

$$M_{med} \approx n\cdot(\ln m + \gamma + 1) + 1 = O(n\cdot \ln n)$$

- În concluzie, algoritmul bazat pe selecție este de **preferat** celui bazat pe inserție, cu toate că în cazurile în care cheile sunt ordonate, sau aproape ordonate, sortarea prin inserție este mai rapidă.

3.2.3. Sortarea prin interschimbare

- Clasificarea metodelor de sortare în diferite familii ca **inserție**, **interschimbare** sau **selecție** nu este întotdeauna foarte bine definită.
- Paragrafele anterioare au analizat algoritmi care deşi implementează inserția sau selecția, se bazează pe fapt pe interschimbare.
- În acest paragraf se prezintă o metodă de sortare în care **interschimbarea** a două elemente este caracteristica dominantă.
- **Principiul** de bază al acestei metode 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 mulțimii rămase spre capătul din stânga al tabloului.

- Dacă se consideră tabloul în poziție verticală şi se asimilează elementele sale cu nişte bule de aer în interiorul unui lichid, fiecare bulă având o **greutate** proporțională cu valoarea cheii, atunci fiecare trecere prin tablou se soldează cu **ascensiunea** unei bule la nivelul specific de greutate.
- Din acest motiv această metodă de sortare este cunoscută în literatură sub denumirea de **bubblesort** adică **sortare prin metoda bulelor**.
- Algoritmul aferent acestei metode apare în continuare [3.2.3.a]:

```
----------------------------------------------------------------
/*Sortarea prin interschimbare Bubblesort - Varianta 1*/

void bubblesort()
{
  tipindice i,j; tipelement temp; tipelement a[n];

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

- Profilul temporal al algoritmului de sortare prin interschimbare este prezentat în figura 3.2.a şi el conduce la o estimare a încadrării performanței algoritmului în ordinul $O(n^2)$.

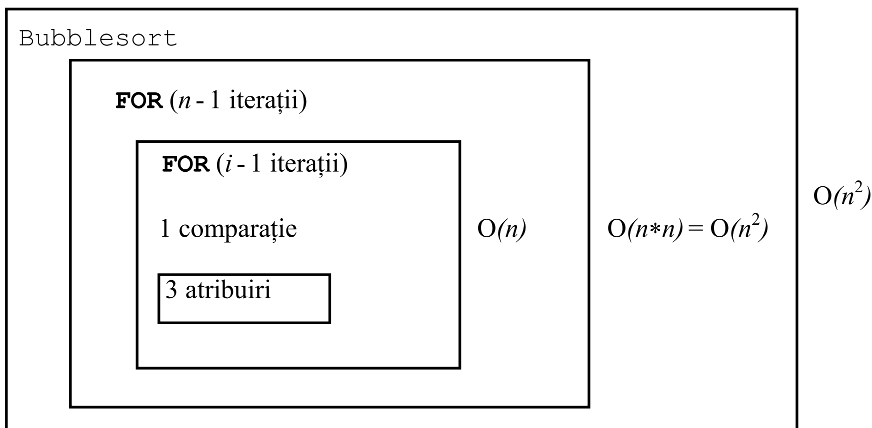


Fig.3.2.a. Profilul temporal al sortării prin interschimbare

- În multe cazuri ordonarea se termină **înainte** de a se parcurge toate iterațiile buclei **FOR** exterioare.
    - În acest caz, restul iterațiilor sunt fără efect, deoarece elementele sunt deja ordonate.
- O modalitate evidentă de **îmbunătățire** a algoritmului bazată pe această observație este aceea prin care se memorează dacă a avut sau nu loc vreo interschimbare în cursul unei treceri.

- Şi în acest caz este însă necesară o ultimă trecere, fără nici o interschimbare care marchează finalizarea algoritmului.

```
------------------------------------------------------------------
/*Sortarea prin interschimbare (varianta 2)*/
typedef int boolean; tipelement a[n];
#define true (1)
#define false (0)

void bubblesort1()
{
    tipindice i; boolean modificat;
    tipelement temp;

  do {
    modificat= false;
    for( i= 0; i < n-1; i ++)
      if (a[i].cheie>a[i+1].cheie)           /*[3.2.3.b]*/
        {
          temp= a[i]; a[i]= a[i+1]; a[i+1]= temp;
          modificat= true;
        }
  } while (modificat);
}
------------------------------------------------------------------
```

- **Analiza sortării bubblesort**

- Numărul comparațiilor la algoritmul bubblesort este constant şi are valoarea:

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

- Valorile minimă, maximă şi medie ale numărului de mişcări sunt:

$$M_{min} = 0$$

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

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

- Analiza comparativă a performanțelor algoritmilor de sortare prezentați, scoate în evidență faptul că **sortarea prin interschimbare** este mai puțin performantă decât sortările prin inserție sau selecție, astfel încât utilizarea ei nu este recomandabilă.
- Se poate demonstra că **distanța medie** pe care fiecare element al unui tablou de dimensiune n, o parcurge în procesul sortării este de $n/3$ locuri.
- Astfel, deoarece în metodele prezentate până acum, fiecare element îşi modifică doar cu un singur loc poziția la fiecare pas elementar, este necesar un număr de treceri proporțional cu n^2.

* O îmbunătățire efectivă a performanței trebuie să aibă în vedere deplasarea elementelor pe distanțe mai mari într-un singur pas.

3.2.4. Sortarea prin metoda ansamblelor

* Metoda sortării prin selecție se bazează pe selecția repetată a celei mai mici chei dintre n elemente, apoi dintre cele n-1 rămase, etc.
* Este evident că determinarea celei mai mici chei dintre n elemente necesită n-1 comparații, dintre n-1 elemente necesită n-2 comparații etc.
* Activitatea de selecție poate fi **îmbunătățită**, dacă la fiecare trecere se vor reține mai multe informații şi nu doar elementul cu cheia cea mai mică.
    * Astfel spre exemplu din $n/2$ comparații se poate determina cea mai mică cheie a **fiecărei perechi** de elemente,
    * Din alte $n/4$ comparații, cea mai mică cheie a **fiecărei perechi de chei mici** deja determinate şi aşa mai departe.
    * În final, utilizând doar $n/2 + n/4 + \cdots + 4 + 2 + 1 = n-1$ comparații, se poate construi un **arbore de selecții** având drept rădăcină cheia cea mai mică (fig.3.2.5.a).
    * Arborele de selecții este de fapt un **arbore binar parțial ordonat**.

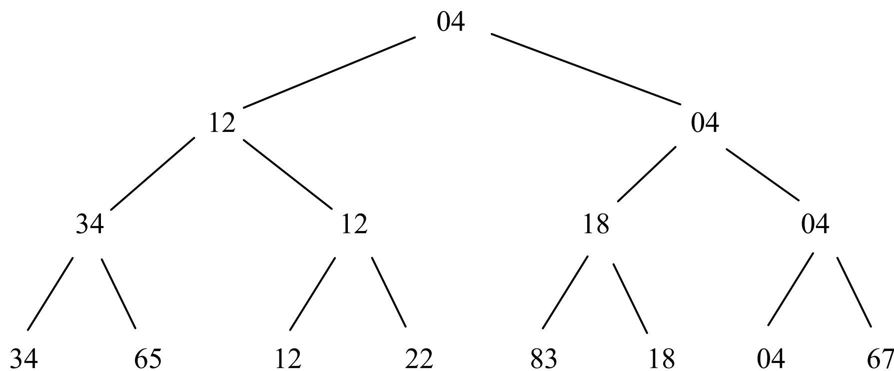


Fig.3.2.5.a. Arbore de selecții

* Cum se poate utiliza acest arbore la sortare?
    * Se extrage cheia cea mai **mică** din rădăcina arborelui.
    * În continuare se parcurge în **sens invers** drumul urmat de cheia cea mai mică şi se elimină succesiv această cheie înlocuind-o:
        * (1) Fie cu un **loc liber**, la baza structurii arbore
        * (2) Fie cu **elementul ramurii alternative** în cazul unui nod intermediar

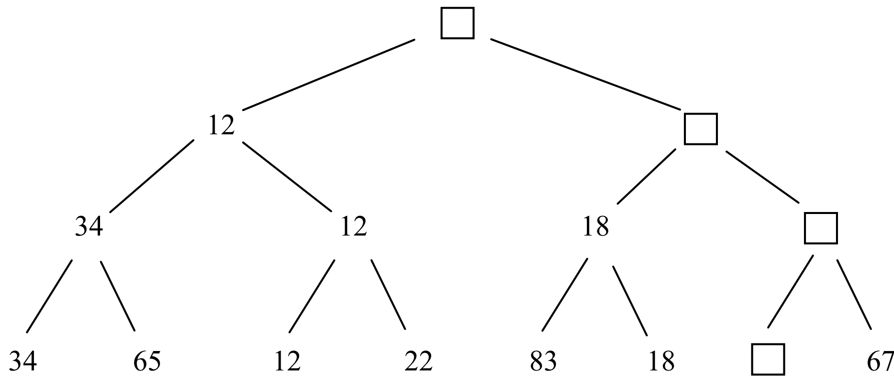


Fig. 3.2.5.b. Selecția celei mai mici chei

- Din nou, elementul care va răzbate spre rădăcina arborelui va fi cel cu cheia cea mai mică (al doilea după cel anterior), element care poate fi extras.

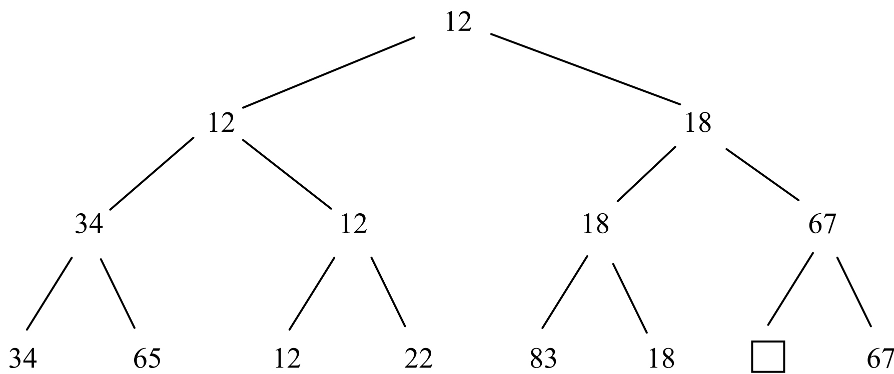


Fig. 3.2.5.c. Completarea locurilor libere

- După `n` astfel de paşi de selecție, s-au extras succesiv cele `n` elemente ale mulțimii în ordine crescătoare, arborele devine vid şi procesul de sortare este încheiat.
- Trebuie notat faptul că fiecare din cei `n` paşi de selecție necesită **numai** $\log_2 n$ comparații.
- În consecință, procesul de sortare integrală necesită:
    - n paşi pentru construcția arborelui,
    - Un număr de operații elementare de ordinul lui $n \cdot \log_2 n$ pentru sortarea propriu-zisă.
- Aceasta este o îmbunătățire considerabilă față de metodele directe care necesită un efort de ordinul $O(n^2)$ şi chiar față de Shellsort care necesită $O(n^{1.2})$.
- Este evident faptul că în cazul metodei de sortare bazată pe structura arbore, **complexitatea** paşilor de sortare individuali creşte.

- De asemenea, în vederea reținerii unei cantități sporite de informație, trebuie concepută o **structură de date** aparte care să permită organizarea eficientă a informației.
    - (1) În primul rând trebuiesc eliminate **locurile goale**, care pe de o parte sporesc dimensiunea arborelui, iar pe de altă parte sunt sursa unor comparații care nu sunt necesare.
    - (2) În al doilea rând, arborele ar trebui reprezentat utilizând locații de memorie pentru `n` elemente şi nu pentru `2n -1` elemente aşa cum rezultă din figurile 3.2.5.a, b, c.
- Aceste probleme au fost rezolvate de către **J. Williams**, creatorul metodei de sortare **heapsort** (*sortare de ansamble*).
- Metoda în sine, reprezintă o realizare de **excepție** printre metodele convenționale de sortare şi utilizează o reprezentare specială a unui arbore binar parțial ordonat, numită ***"ansamblu"***.
- Un **ansamblu** ("**heap**") este definit ca o secvență de chei $h_s, h_{s+1}, ..., h_d$ care se bucură de proprietățile [3.2.5.a]:

$$h_i \le h_{2i} \qquad \text{pentru toți} \quad i = s, ..., d/2 \qquad [3.2.5.a]$$
$$h_i \le h_{2i+1}$$

- Un ansamblu poate fi asimilat cu un **arbore binar parțial ordonat** şi reprezentat printr-un tablou.
- Spre exemplu, ansamblul $h_1, h_2, ..., h_{15}$ poate fi asimilat cu arborele binar din figura 3.2.5.d şi poate fi reprezentat prin tabloul `h` în baza următoarei tehnici:
    - Se **numerotează** elementele ansamblului, nivel cu nivel, de sus în jos, de la stânga la dreapta;
    - Se **asociază** elementelor ansamblului, locațiile unui tablou de elemente `h`, astfel încât elementului h_i al ansamblului îi corespunde locația `h[i]`.

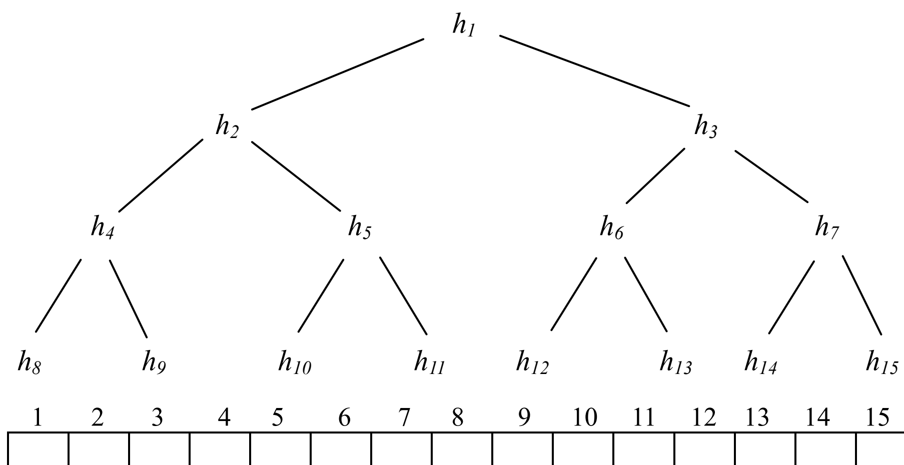


Fig. 3.2.5.d. Reprezentarea unui ansamblu printr-un tablou liniar `h`

- Un ansamblu se bucură de proprietatea că **primul** său element este cel mai mic dintre toate elementele ansamblului adică $h_1 = min(h_1, ..., h_n)$.
- Se presupune un ansamblu $h_{s+1}, h_{s+2}, ..., h_d$ definit prin indicii $s+1$ şi d

- Acestui ansamblu i se adaugă la stânga, pe poziția h_s un nou element x, obținându-se un ansamblu extins spre stânga h_s, ..., h_d.

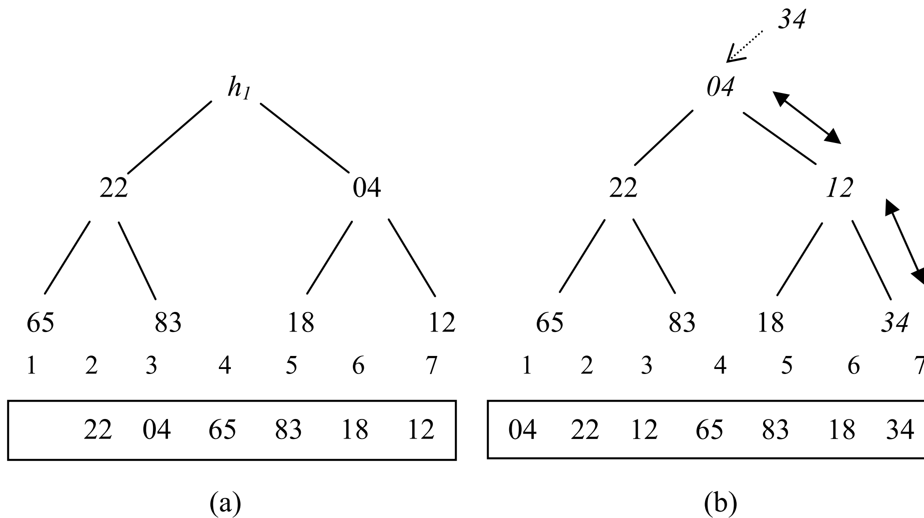


Fig. 3.2.5.e. Deplasarea unei chei într-un ansamblu

- În figura 3.2.5.e.(a) apare ca exemplu ansamblul h_2, ..., h_7, iar în aceeaşi figură (b), ansamblul extins spre stânga cu un element $x = 34$.
    - Noul ansamblu se obține din cel anterior plasând pe x în vârful ansamblului şi deplasându-l "*în jos*" de-a lungul drumului indicat de componentele cele mai mici, care în acelaşi timp urcă.
    - Astfel valoarea 34 este mai întâi schimbată cu valoarea 04, apoi cu valoarea 12, generând structura din figura amintită.
    - Această deplasare conservă condițiile care definesc un ansamblu [3.2.5.a].
- Notând cu `i` şi `j` indicii elementelor care se interschimbă, şi presupunând că x a fost introdus pe poziția h_s, tehnica de implementare a unei astfel de deplasări apare în [3.2.5.b] în variantă pseudocod.

```
----------------------------------------------------------------
*Deplasarea unei chei de sus în jos într-un ansamblu
procedure Deplasare(s,d: TipIndice){limitele ansamblului}
  i:= s;   {indică elementul curent)
  j:= 2*i; {indică fiul stâng al elementului curent}
  temp:= h[i] {elementul care se deplasează}
  cât timp există niveluri în ansamblu (j≤i) şi locul de
             plasare nu a fost găsit execută
    *selectează pe cel mai mic dintre fii elementului
      indicat de i (pe h[j] sau pe h[j+1]
    dacă temp>fiul_selectat atunci
        *deplasează fiul selectat în locul tatălui
          său (h[i]:=h[j]);
        *avansează pe nivelul următor al ansamblului
         (i:=j; j:=2*i)
      altfel                                        [3.2.5.b]
        retur {locul a fost găsit}
    □
  □
  *plasează temp la locul său în ansamblu (h[i]:=temp);
----------------------------------------------------------------
```

* Procedura care implementează algoritmul de deplasare apare în [3.2.5.c].

```
------------------------------------------------------------------
/* Deplasarea unei chei de sus în jos într-un ansamblu */
void deplasare(tipindice s,tipindice d)
{
  tipindice i,j;
  tipelement temp;
  boolean ret;

  i= s;  j= 2*i;
  temp= h[i];
  ret= false;
  while((j<=d) && (! ret))
    {
      if (j<d)
        if (h[j].cheie>h[j+1].cheie)  j= j+1;
      if (temp.cheie>h[j].cheie)
          {
            h[i]= h[j]; i= j; j= 2*i;         /*[3.2.5.c]*/
          }
        else
          ret= true;
    }
  h[i]= temp;
}
------------------------------------------------------------------
```

* **R.W. Floyd** a sugerat o metodă de a **construi** un ansamblu **in situ**, utilizând procedura de deplasare în ansamblu prezentată mai sus:
    * Se consideră un tablou h_1, ..., h_n în care în mod evident elementele $h_{n/2}$, ..., h_n formează deja un **ansamblu** deoarece **nu** există nici o pereche de indici `i` şi `j` care să satisfacă relaţia `j=2*i` (sau `j=2*i+1`).
    * Aceste elemente formează cea ce poate fi considerat drept **şirul de bază** al ansamblului asociat.
    * În continuare, ansamblul este **extins spre stânga**, la fiecare pas cu câte un element, introdus în vârf şi deplasat până la locul său.
    * Prin urmare, considerând că tabloul iniţial este memorat în `h`, procesul de generare *"in situ"* al unui ansamblu poate fi descris prin secvenţa [3.2.5.d].

```
------------------------------------------------------------------
{Faza creare ansamblu}
s:= (n DIV 2)+1;
WHILE s>1 DO
  BEGIN                                         [3.2.5.d]
    s:= s-1; Deplasare(s,n)
  END;{WHILE}
------------------------------------------------------------------
```

* În vederea **sortării elementelor**, se execută n paşi de `Deplasare`, după fiecare pas selectându-se vârful ansamblului.
* Problema care apare este aceea a **locului** în care se memorează vârfurile consecutive ale ansamblului, respectiv elementele sortate, respectând constrângerea "*in situ*".
* Această problemă poate fi rezolvată astfel:
    * În fiecare pas al procesului de sortare **se interschimbă** ultima componentă a ansamblului cu componenta aflată în vârful acestuia (`h[1]`).

* După fiecare astfel de interschimbare ansamblul **se restrânge** la dreapta cu o componentă.

* În continuare se lasă componenta din vârf (`h[1]`) să se **deplaseze** spre locul său în ansamblu.

* În termenii procedurii `Deplasare` această tehnică poate fi descrisă ca în secvența [3.2.5.e].

```
----------------------------------------------------------------
{Faza sortare}
d:= n;
WHILE d >1 DO
  BEGIN
    temp:= h[1]; h[1]:= h[d]; h[d]:= temp;      [3.2.5.e]
    d:= d-1; Deplasare(1,d)
  END;{WHILE}
----------------------------------------------------------------
```

* Cheile se obțin sortate în **ordine inversă**, lucru care poate fi ușor remediat modificând sensul relațiilor de comparație din cadrul procedurii `Deplasare`.

* Rezultă următorul algoritm care ilustrează **tehnica de sortare heapsort** [3.2.5.f].

```
----------------------------------------------------------------
/* Sortare prin metoda ansamblelor - Heapsort */

void heapsort();

static void deplasare1(tipindice* s, tipelement* temp,
tipindice* d)
  {
    tipindice i,j; boolean ret;

    i=*s; j= 2*i; *temp= h[i]; ret= false;
    while ((j<=*d) && (! ret))
      {
        if (j<*d)
          if (h[j].cheie<h[j+1].cheie)  j= j+1;
        if (temp->cheie<h[j])
            {
              h[i]= h[j]; i= j; j= 2*i;
            }
          else
            ret= true;
      }   /*WHILE*/
    h[i]= *temp;
  }      /*Deplasare*/

void heapsort()
{
    tipindice s,d; tipelement temp;

      /*constructie ansamblu*/
  s= (n / 2)+1; d= n;                          /*[3.2.5.f]*/
  while (s>1)
    {
      s= s-1; deplasare1(&s, &temp, &d);
    }   /*WHILE*/

  while (d>1)    /*sortare*/
    {
```

```
      temp= h[1]; h[1]= h[d]; h[d]= temp;
      d= d-1; deplasare1(&s, &temp, &d);
    }
}
```

• Analiza metodei heapsort

- La prima vedere nu rezultă în mod evident faptul că această metodă conduce la rezultate bune.
- Analiza performanţelor metodei heapsort contrazice însă această părere.
- (1) La faza de **construcţie a ansamblului** sunt necesari $n/2$ paşi de deplasare,
    - În fiecare fiecare pas se mută elemente de-a lungul a respectiv $log(n/2)$, $log(n/2+1)$, ..., $log(n\text{-}1)$ poziţii, (în cel mai defavorabil caz), unde logaritmul se ia în baza 2 şi se trunchiază la prima valoare întreagă.
- (2) În continuare, faza de **sortare** necesită n-1 deplasări cu cel mult $log\ (n\text{-}2)$, $log\ (n\text{-}1)$, ..., 1 mişcări.
- (3) În plus mai sunt necesare $3 \cdot (n\text{-}1)$ mişcări pentru a **aşeza** elementele sortate în ordine.
- Toate acestea dovedesc că în cel mai defavorabil caz, tehnica **heapsort** are nevoie de un număr de paşi de ordinul $O(n \cdot log\ n)$ [3.2.5.g].

$$O(n/2 \cdot log_2(n\text{-}1) + (n\text{-}1) \cdot log_2(n\text{-}1) + 3 \cdot (n\text{-}1)) = O(n \cdot log_2 n) \qquad [3.2.5.g]$$

- Este greu de determinat cazul cel mai defavorabil şi cazul cel mai favorabil pentru această metodă.
- Numărul mediu de **mişcări** este aproximativ egal cu $1/2 \cdot n \cdot log\ n$, deviaţiile de la această valoare fiind relativ mici.
- În manieră specifică metodelor de sortare avansate, valorile mici ale numărului de elemente `n`, **nu** sunt suficient de reprezentative, eficienţa metodei crescând o dată cu creşterea lui `n`.

3.2.5. Sortarea prin partiţionare

- Deşi metoda **bubblesort** bazată pe principiul interschimbării este cea mai puţin performantă dintre metodele studiate,
- **C.A.R. Hoare**, pornind de la acelaşi principiu, a conceput o metodă cu performanţe spectaculare pe care a denumit-o **quicksort** (sortare rapidă).
- Aceasta se bazează pe aceeaşi idee de a creşte eficienţa interschimbărilor prin mărirea distanţei dintre elementele implicate.
- Sortarea prin partiţionare porneşte de la următorul **algoritm**.
    - Fie x un **element oarecare** al tabloului de sortat a_1, ..., a_n.
    - Se parcurge tabloul de la **stânga spre dreapta** până se găseşte primul element $a_i > x$.
    - În continuare se parcurge tabloul de la **dreapta spre stânga** până se găseşte primul element $a_j < x$.
    - Se **interschimbă** între ele elementele a_i şi a_j,

- Se continuă parcurgerea tabloului de la stânga respectiv de la dreapta (din punctele în care s-a ajuns anterior), până se găsesc alte două elemente care se interschimbă, ş.a.m.d.

- Procesul se termină când cele două parcurgeri se *"întâlnesc"* undeva în interiorul tabloului.

- Efectul final este că acum şirul iniţial este **partiţionat** într-o **partiţie stânga** cu chei mai mici decât *x* şi o **partiţie dreapta** cu chei mai mari decât *x*.

- Considerând elementele şirului memorate în tabloul `a`, principiul partiţionării apare prezentat sintetic în [3.2.6.a].

```
----------------------------------------------------------------
*Partiţionarea unui tablou - varianta pseudocod

procedure Partiţionare   {Partiţionează tabloul a[s..d]}
*selectează elementul x (de regulă de la mijlocul
                         intervalului de partiţionat)
 repetă
   *caută primul element a[i]>x, parcurgând
      intervalul de la stânga la dreapta
   *caută primul element a[j]<x, parcurgând
      intervalul de la dreapta la stânga
   dacă i<=j  atunci                         [3.2.6.a]
     *interschimbă pe a[i] cu a[j]
   □
 până când  parcurgerile se întâlnesc (i>j)
 □
-----------------------------------------------------------
```

- Înainte de a trece la sortarea propriu-zisă, se dă o formulare mai precisă partiţionării în forma unei proceduri [3.2.6.b].

- Se precizează că relaţiile > respectiv < au fost înlocuite cu ≥ respectiv ≤ ale căror negate utilizate în instrucţiunile **`WHILE`** sunt < respectiv >.
    - În acest caz `x` joacă rol de **fanion** pentru ambele parcurgeri.

```
------------------------------------------------------------
*/{Procedura Partiţionare

void partitionare()
 {
    tipelement x,temp;

    i= 1; j= n;
    x= a[n / 2];                         /*[3.2.6.b]*/
    do {
      while (a[i].cheie<x.cheie)  i= i+1;
      while (a[j].cheie>x.cheie)  j= j-1;
      if (i<=j)
        {
        temp= a[i]; a[i]= a[j]; a[j]= temp;
        i= i+1; j= j-1;
         }
     } while (!(i>j));
 }
--------------------------------------------------------
```

- În continuare, cu ajutorul partiţionării, sortarea se realizează simplu:
    - După o primă partiţionare a secvenţei de elemente se aplică aceeaşi procedură celor două partiţii rezultate,

- Apoi celor patru partiții ale acestora, ş.a.m.d.
- Procesul se termină când fiecare partiție se reduce la un singur element.

- Tehnica sortării bazată pe partiționare este ilustrată în secvența [3.2.6.c].

```
------------------------------------------------------------
*Sortarea prin partiționare -quicksort - varianta pseudocod

procedure QuickSort(s,d);
  *partiționează intervalul s,d față de Mijloc
  dacă există partiție stânga atunci
    QuickSort(s,Mijloc-1)                       [3.2.6.c]
  dacă există partiție dreapta atunci
    QuickSort(Mijloc+1,d);
--------------------------------------------------------
```

- În secvența [3.2.6.e] apare o variantă de implementare a sortării quicksort

```
------------------------------------------------------------
//Sortarea prin partiționare - quicksort

quicksort(int s,int d) { //int a[],int n
 int i=s,j=d,x=a[(s+d)/2];
 do {
  while(a[i]<x)i++;
  while(a[j]>x)j--;
  if(i<=j) {
   int temp=a[i];                                  [3.2.6.e]
   a[i]=a[j];
   a[j]=temp;
   i++;j--;
  }
 }while(i<=j);
 if(s<j)quicksort(s,j);
 if(d>i)quicksort(i,d);
}
------------------------------------------------------------
```

- ## Analiza metodei quicksort

- Pentru a analiza performanța acestei metode, se analizează mai întâi partiționarea.
  - Se presupun pentru simplificare următoarele precondiții:
    - (1) Setul ce urmează a fi partiționat constă din n chei **distincte** şi **unice** cu valorile $\{1, 2, 3, ..., n\}$
    - (2) Dintre cele n chei a fost selectată cheia cu valoarea x în vederea partiționării.
  - În consecință această cheie ocupă a ***x*-a poziție** în mulțimea ordonată a cheilor şi ea poartă denumirea de **pivot**.
- Se ridică următoarele întrebări:

1. Care este **probabilitatea** ca după ce a fost selectată cheia cu **valoarea** x ca pivot, o cheie oarecare a partiției să fie **interschimbată** ?
    - Pentru ca o cheie să fie interschimbată ea trebuie să fie mai mare ca x.
    - Sunt n-x+1 chei mai mari ca x
    - Rezultă că probabilitatea ca o cheie oarecare să fie interschimbată este $(n - x + 1) / n$ (raportul dintre numărul de cazuri favorabile şi numărul de cazuri posibile).

2. Care este **numărul de interschimbări** necesar la o partiționare a *n* chei pentru care s-a selectat ca pivot cheia situată pe **poziția** *x* ?
    - La dreapta pivotului există $n-x$ poziții care vor fi procesate în procesul de partiționare.
    - **Numărul posibil de interschimbări** în acest context este deci egal cu produsul dintre numărul de chei care vor fi selectate ($n-x$) şi probabilitatea ca o cheie selectată să fie interschimbată.

$$NrInt = (n-x) \cdot \frac{(n-x+1)}{n}$$

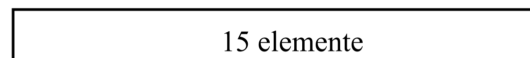
3. Care este numărul **mediu de interschimbări** pentru partiționarea unei **secvențe de *n* chei**?
    - La o partiționare poate fi selectată oricare din cele *n* chei ca şi pivot, deci *x* poate lua orice valoare cuprinsă între 1 şi *n*.
    - **Numărul mediu** M de interschimbări pentru partiționarea **unei** secvențe de *n* chei se obține însumând toate numerele de interschimbări pentru toate valorile lui *x* cuprinse între 1 şi *n* şi împărțind la numărul total de chei *n* [3.2.6.h].

$$\text{M} = \frac{1}{n} \cdot \sum_{x=1}^{n} NrInt = \frac{1}{n} \cdot \sum_{x=1}^{n} (n-x) \cdot \frac{(n-x+1)}{n} = \frac{6}{n} - \frac{1}{6 \cdot n} \approx \frac{n}{6} \qquad [3.2.6.h]$$

- Presupunând în mod exagerat că întotdeauna va fi selectată **mediana** partiției (mijlocul său valoric), fiecare partiționare va divide tabloul în două jumătăți egale.
- Se face precizarea că **mediana** este elementul situat ca şi valoare în mijlocul partiției, dacă aceasta este ordonată.
- În aceste condiții este necesar un număr de treceri prin toate elementele tabloului egal cu *log n* (fig.3.2.6.a).
- După cum rezultă din figura 3.2.6.a, pentru un tablou de 15 elemente sunt necesare $\lceil log_2 15 \rceil = 4$ treceri prin toate elementele tabloului sau 4 paşi de partiționare integrală a tabloului.

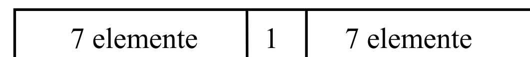
Număr apeluri ***Număr***
elemente de
partiționat pe
apel

1 apel sortare
(15 elemente)



15

2 apeluri sortare
(7 elemente)



7

4 apeluri sortare
(3 elemente)

3

8 apeluri sortare
(1 element)

1

Fig. 3.2.6.a. Funcționarea principială a sortării prin partiționare

- Din păcate numărul de apeluri recursive ale procedurii este egal cu 15, adică exact cu numărul de elemente.
- Rezultă că numărul total de **comparații** este $n \cdot log\ n$ (la o trecere sunt comparate toate cheile)
- Numărul de **mişcări** este $n/6 \cdot log\ n$ deoarece conform formulei [3.2.6.h] la partiționarea a n chei sunt necesare în medie $n/6$ mişcări [3.2.6.i].

$$C = n \cdot log_2\ n \qquad M = \frac{1}{6} \cdot n \cdot log_2\ n \qquad [3.2.6.i]$$

- Aceste rezultate sunt **excepțional** de bune, dar se referă numai la **cazul optim** în care s-a presupus că la fiecare trecere se selectează mediana, eveniment care de altfel are probabilitatea doar $1/n$.
- Marele succes al algoritmului quicksort se datoreşte însă faptului surprinzător că **performanța sa medie**, la care alegerea pivotului se face la întâmplare, este inferioară performanței optime doar cu un factor egal cu $2 \cdot ln(2) =$ 1.4 deci cu aproximativ 40 % [Wi76].
- Tehnica prezentată are însă şi **dezavantaje**.
  - În primul rând ca şi la toate metodele de sortare avansate, performanțele ei sunt **moderate** pentru valori mici ale lui n.
  - Un al doilea dezavantaj, se referă la **cazul cel mai defavorabil** în care performanța metodei scade catastrofal.
    - Acest caz apare când la fiecare partiționare este selectată cea mai mare (cea mai mică) valoare ca şi pivot.
    - Fiecare pas va partaja în acest caz secvența formată din n elemente, într-o partiție stânga cu $n-1$ elemente şi o partiție dreapta cu un singur element.
    - Vor fi necesare astfel n partiționări în loc de $\log(n)$, iar performanța obține valori de ordinul $O(n^2)$.
- Tehnica quicksort se comportă straniu:
  - Are **performanțe slabe** în cazul sortărilor banale
  - Are **performanțe deosebite** în cazul tablourilor dezordonate
- De asemenea, dacă x se alege întotdeauna la mijloc (mediana), atunci **tabloul sortat invers** devine **cazul optim** al sortării quicksort.

3.2.6. Determinarea medianei

- Mediana a n elemente este definită ca fiind acel element care este mai mic (sau egal) decât **jumătate** din elemente şi este mai mare (sau egal) decât cealaltă jumătate.
  - Spre exemplu mediana secvenței 16, 12, 99, 95, 18, 87, 10 este 18.
- Problema aflării medianei este corelată direct cu cea a **sortării** deoarece, o metodă sigură de a determina mediana este următoarea:
  - (1) Se **sortează** cele n elemente
  - (2) Se **extrage** elementul din mijloc.
- Tehnica **partiționării** poate însă conduce la o metodă generală mai rapidă, cu ajutorul căreia se poate determina cel de-al ***k*-lea element** ca valoare dintre n elemente.
  - Găsirea medianei reprezintă cazul special $k = n/2$.

- o În acelaşi context, $k=1$ precizează aflarea **minimului**, iar $k=n$, aflarea **maximului**.

- Algoritmul conceput de C.A.R. **Hoare** funcționează după cum urmează.
  - o Se presupune că elementele avute în vedere sunt memorate în tabloul `a` cu dimensiunea `n`,
  - o Pentru început se realizează o partiționare cu limitele `s = 0`, `d = n+1` şi cu `a[k]` selectat pe post de pivot `x`.
  - o În urma acestei partiționări rezultă valorile index `i` şi `j` care satisfac relațiile [3.2.7.a].

```
1)  x = a[k]
2)  a[h] ≤ x   pentru toți  h < i
3)  a[h] ≥ x   pentru toți  h > j                    [3.2.7.a]
4)  i > j
```

- o Sunt posibile trei situații:
  - (1) Valoarea pivotului `x` a fost prea **mică**, astfel încât limita dintre cele două partiții este sub valoarea dorită `k`.
    - Procesul de partiționare se reia pentru elementele `a[i],...,a[d]` (partiția dreapta) (fig.3.2.7.a (a)).
  - (2) Valoarea pivotului `x` a fost prea **mare**.
    - Operația de partiționare se reia pentru partiția `a[s],...,a[j]` (partiția stânga) (fig.3.2.7.a (b)).
  - (3) `j < k < i`.
    - În acest caz elementul `a[k]` separă tabloul în două partiții, el desemnând **mediana** (fig.3.2.7.a (c)).
- o Procesul de partiționare se repetă până la realizarea cazului (3).

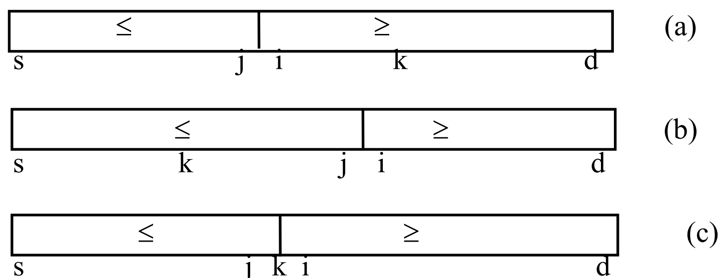


Fig. 3.2.7.a. Determinarea medianei

- Algoritmul aferent este prezentat în variantă pseudocod în secvența [3.2.7.b] respectiv o primă rafinare în secvența de program [3.2.7.c].

```
*Aflarea medianei - Varianta pseudocod - Pas rafinare 0

  procedure Mediana (s,d,k);
    cât timp există partiție                       [3.2.7.b]
       *alege pivotul (elementul din poziția k)
       *partiționează intervalul curent fața de valoarea
```

```
         pivotului
       dacă poziție pivot<k atunci *selectează partiția
           dreapta
       dacă poziție pivot>k atunci *selectează partiția
           stânga
   □
------------------------------------------------------------
{Procedura Mediana - Pas rafinare 1}
s:= 0; d:= n-1 ;
WHILE s<d DO
  BEGIN
    x:= a[k-1];                                    [3.2.7.c]
    *se partiționează  a[s]...a[d]
    IF j<k THEN s:= i;
    IF k<i THEN d:= j
  END;
------------------------------------------------------------
```

- Programul aferent apare în secvența [3.2.7.d]

```
------------------------------------------------------------
/* Procedura Mediana -  */

void mediana (int k)
  {
      tipindice s,d,i,j; tipelement x,temp;

    s=0; d=n-1;
    while (s<d)
      {
        x= a[k-1]; i= s; j= d;
        do {   /*partitionarea*/             /*[3.2.7.d]*/
          while (x<a[i])
                i++;
          while (x>a[j])
                j--;
          if (i<=j)
           {
              temp= a[i]; a[i]= a[j]; a[j]= temp;
              i++; j--;
            }
        } while (!(i>j));
        if (j<k)  s= i;
        if (k<i)  d= j;
      }
  }
----------------------------------------------------------------
```

- Dacă se presupune că în medie fiecare partiționare înjumătățeşte partiția în care se găseşte elementul căutat, atunci numărul necesar de comparații C este de ordinul O(n) [3.2.7.e].

$$C = n + \frac{n}{2} + \frac{n}{4} + \cdots + 1 = 2 \cdot n - 1 \qquad [3.2.7.e]$$

- Numărul de mişcări M **nu** poate depăşi numărul de comparații, el fiind de regulă mai mic, ca atare tot O(n)

* Valorile indicatorilor C şi M estimaţi pentru **determinarea medianei** subliniază superioritatea acestei metode
    * În acelaşi timp explică performanţa metodei faţă de metodele bazate pe sortarea tabloului şi extragerea celui de-al *k*-lea element, a căror performanţă în cel mai bun caz este de ordinul $O(n \cdot log\ n)$.

3.2.7. Sortarea binsort. Determinarea distribuţiei cheilor

* În general algoritmii de sortare bazaţi pe metode avansate au nevoie de $O(n \cdot log\ n)$ paşi pentru a sorta *n* elemente.
* Trebuie precizat însă faptul că acest lucru este valabil în situaţia în care:
    * **Nu** există nici o altă informaţie suplimentară referitoare la chei, decât faptul că pe mulţimea acestora este definită o **relaţie de ordonare**, prin intermediul căreia se poate preciza dacă valoarea unei chei este **mai mică** respectiv **mai mare** decât o alta.
* După cum se va vedea în continuare, sortarea se poate face şi **mai rapid** decât în limitele performanţei $O(n \cdot log\ n)$, **dacă**:
    * (1) Există şi **alte informaţii referitoare** la cheile care urmează a fi sortate
    * (2) Se **renunţă** măcar parţial la constrângerea de sortare ***"in situ"***
* Spre **exemplu**,
    * Se cere să se sorteze un set de *n* chei de tip întreg, ale căror valori sunt unice şi aparţin intervalului de la 1 la *n*.
    * Dacă `a` şi `b` sunt tablouri identice cu câte *n* elemente, `a` conţinând cheile care urmează a fi sortate, atunci sortarea se poate realiza direct în tabloul `b` conform secvenţei [3.2.8.a].

```
------------------------------------------------------------
Exemplu de sortare liniară
     FOR i:= 1 TO n DO
       b[a[i].cheie]:= a[i];     {O(n)}            [3.2.8.a]
------------------------------------------------------------
```

* Ideea metodei:
    * Pentru fiecare element `a[i]` se determină locul elementului şi se plasează elementul la locul potrivit în tabloul `b`
    * Întregul ciclu necesită $O(n)$ paşi.
    * Rezultatul este însă corect numai în cazul în care există **un singur element** cu cheia `x`, pentru fiecare valoare cuprinsă între `[1,n]`.
        * Un al doilea element cu aceeaşi cheie va fi introdus tot în `b[x]` distrugând elementul anterior.
* Secvenţa [3.2.8.a] ilustrează tehnica de sortare numită **binsort**, în cadrul căreia se crează **bin**-uri, fiecare bin păstrând un element sortat cu o anumită cheie [AHU85].
* Tehnica sortării este simplă:
    * (1) Se examinează fiecare element de sortat
    * (2) Se introduce în **bin**-ul corespunzător valorii cheii.
        * În secvenţa [3.2.8.a] bin-urile sunt chiar elementele tabloului `b`, unde `b[i]` este binul cheii având valoarea `I`
* Tehnica aceasta simplă şi de performantă se bazează pe următoarele **cerinţe apriorice**:
    * (1) **Domeniul limitat** al cheilor (1, *n*)

- (2) **Unicitatea** fiecărei chei.
- Dacă cea de-a doua cerință **nu** este respectată, şi de fapt acesta este cazul obişnuit, este necesar ca într-un bin să fie memorate **mai multe elemente** având aceeaşi cheie.
  - Acest lucru se realizează fie prin înşiruire, fie prin concatenare, fiind utilizate în acest scop structuri de date listă.
  - Această situație **nu** deteriorează prea mult performanțele acestei tehnici, efortul de sortare ajungând egal cu O*(n+m)*, unde *n* este numărul de elemente iar *m* numărul de chei,
  - Din acest motiv, această metodă reprezintă punctul de plecare al mai multor tehnici de sortare a structurilor listă [AHU85].
- Spre exemplu, o metodă de rezolvare a unei astfel de situații este cea bazată pe **determinarea distribuției cheilor** (**"distribution counting"**) [Se88].
- **Problema** se formulează astfel:
  - Se cere să se sorteze un tablou cu `n` articole ale căror chei sunt cuprinse între 0 şi *m*-1.
  - Dacă *m* **nu** este prea mare pentru rezolvarea problemei poate fi utilizat algoritmul de ***"determinare a distribuției cheilor"***.
- **Ideea** algoritmului este următoarea:
  - Se contorizează într-o primă trecere numărul de chei pentru fiecare valoare de cheie care apare în tabloul `a`;
  - Se ajustează valorile contoarelor;
  - Într-o a doua trecere, utilizând aceste contoare, se mută direct articolele în poziția lor ordonată în tabloul `b`.
- Formularea algoritmului este cea din secvența [3.2.8.c].
  - Pentru simplificare se presupune că tabloul `a` este un tablou care conține doar chei.

```
------------------------------------------------------------
/* Sortare cu determinarea distribuţiilor cheilor - */
enum {n = 10, m = 10};

typedef unsigned tipcheie;
typedef unsigned tipindice;

typedef tipcheie tiptablou[n];
tipcheie numar[m];
tiptablou a,b;
tipindice i,j;

int main(int argc, const char* argv[])
{
  for( j= 1; j <= m-1; j ++) numar[j]= 0;
  for( i= 1; i <= n;   i ++) numar[a[i-1]]= numar[a[i-1]]+1;
  for( j= 1; j <= m-1; j ++) numar[j]= numar[j-1]+numar[j];
  for( i=n; i >= 1; i --)
    {
      b[numar[a[i-1]]-1]= a[i-1];                  /*[3.2.8.c]*/
      numar[a[i-1]]= numar[a[i-1]]-1;
    }
  for( i= 1; i <= n; i ++) a[i-1]= b[i-1];
  return 0;
}
----------------------------------------------------------------
```

- Contoarele asociate cheilor sunt memorate în tabloul `numar` de dimensiune `m`.
- Inițial locațiile sunt inițializate pe zero (prima buclă `for`)
- Sunt contorizate cheile (a doua buclă `for`).
- În continuare sunt ajustate valorile contoarelor tabloului `numar` (a treia buclă `for`)
- Se parcurge tabloul `a` de la sfârşit spre început, iar cheile sunt introduse exact la locul lor în tabloul `b` cu ajutorul contoarelor memorate în tabloul `numar` (a patra buclă `for`).
    - Concomitent cu introducerea cheilor are loc şi decrementarea contoarelor specifice astfel încât în final, cheile identice să apară în binul specific în ordinea relativă în care apar ele în secvența inițială.
- Ultima buclă realizează mutarea integrală a elementelor tabloului `b` în tabloul `a`, dacă acest lucru este necesar.
- Deşi se realizează mai multe treceri prin elementele tabloului totuşi în ansamblu, performanța algoritmului este O*(n)* .
- Aceasta metodă de sortare pe lângă faptul că este rapidă are avantajul de a fi **stabilă**, motiv pentru care ea stă la baza mai multor metode de sortare de tip **radix.**
- În continuare se prezintă un exemplu de funcționare a algoritmului de sortare bazat pe determinarea distribuției cheilor.

- **Exemplul 3.2.8.** Schematic, în vederea sortării cu determinarea distribuției cheilor se parcurg următorii paşi.

1) Se consideră inițial că tabloul `a` are următorul conținut:

1	2	3	4	5	6	7	8	9	10	11	12	13	14
a	b	b	a	c	a	d	a	b	b	a	d	d	a
0	1	1	0	2	0	3	0	1	1	0	3	3	0

2) Se inițializează tabloul `numar`

0	1	2	3
0	0	0	0

3) Se contorizează valorile cheilor tabloului `a`

0	1	2	3
6	4	1	3

4) Se ajustează valorile tabloului `numar`

0	1	2	3
6	10	11	14

5) Se iau elementele tabloului `a` de la dreapta la stânga şi se introduc pe rând în tabloul `b,` fiecare în poziția indicată de contorul propriu din tabloul `numar`.

    - După introducerea fiecărui element în tabloul `b`, contorul specific din tabloul `numar` este decrementat cu o unitate

1	2	3	4	5	6	7	8	9	10	11	12	13	14
a	a	a	a	a	a	b	b	b	b	c	d	d	d

3.2.8. Sortarea tablourilor cu articole de mari dimensiuni. Sortarea indirectă

- În situația în care tablourile de sortat au **elemente de mari dimensiuni,** regia mutării acestor elemente în procesul sortării este mare.
- De aceea este mult mai convenabil ca:
  - (1) Algoritmul de sortare să opereze **indirect** asupra tabloului original prin intermediul unui **tablou de indici**
  - (2) Tabloul original să fie sortat ulterior într-o singură trecere.
- Ideea metodei:
  - Se consideră un tablou `a[1..n]` cu elemente de mari dimensiuni
  - Se asociază lui `a` un tablou de indici (indicatori) `p[1..n];`
  - Inițial se completează `p[i]:=i` pentru `i=1,n;`
  - Algoritmul utilizat în sortare se modifică astfel încât să se acceseze elementele tabloului `a` prin construcția `a[p[i]]` în loc de `a[i]`.
  - Accesul la `a[i]` prin `p[i]` se va realiza numai pentru comparații, mutările efectuându-se în tabloul `p[i]`.
  - Cu alte cuvinte algoritmul va **sorta** tabloul de indici astfel încât `p[1]` va conține indicele celui mai mic element al tabloului `a`, `p[2]` indicele elementului următor, etc.
- În acest mod se **evită** regia mutării unor elemente de mari dimensiuni.
- Se realizează de fapt o **sortare indirectă** a tabloului `a`.
- Principial o astfel de sortare este prezentată în figura 3.2.10.

Tabloul `a` **înainte** de sortare:

1	2	3	4	5	6	7	8	9	10
32	22	0	1	5	16	99	4	3	50

Tabloul de indici `p` **înainte** de sortare:

1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10

Tabloul `a` **după** sortare:

1	2	3	4	5	6	7	8	9	10
32	22	0	1	5	16	99	4	3	50

Tabloul de indici `p` **după** sortare:

1	2	3	4	5	6	7	8	9	10
3	4	9	8	5	6	2	1	10	7

Fig. 3.2.10. Exemplu de sortare indirectă

- Această idee poate fi aplicată practic oricărui algoritm de sortare.
- Pentru exemplificare în secvența [3.2.10.a] se prezintă un algoritm care realizează **sortarea indirectă** bazată pe **metoda inserției** a unui tablou `a`.

```
-----------------------------------------------------------
/* Sortare indirectă bazată pe metoda inserţiei a unui tablou a
de mari dimensiuni - Varianta C */

tipelement a1[n-0+1];
tipindice p[n-0+1];
void insertieindirecta()
  {
      tipindice i,j,v;
                                                      /*[3.2.10.a]*/
    for( i= 0; i <= n; i ++) p[i]= i;
    for( i= 2; i <= n; i ++)
      {
        v= p[i]; a1[0]= a1[i]; j= i-1;
        while (a1[p[j]].cheie>a1[v].cheie)
          {
            p[j+1]= p[j];
            j= j-1;
          }
        p[j+1]= v;
      }
  }
-----------------------------------------------------------------
```

- După cum se observă, cu excepţia atribuirii fanionului, accesele la tabloul `a` se realizează **numai** pentru comparaţii.
- În multe aplicaţii este suficientă numai obţinerea tabloului `p` nemaifiind necesară şi permutarea elementelor tabloului.
    - Spre exemplu, în procesul tipăririi, elementele pot fi listate în ordine, referirea la ele realizându-se simplu, în mod indirect prin tabloul de indici.
- Dacă este absolut necesară mutarea, cel mai simplu acest lucru se poate realiza într-un alt tablou `b`.
- Dacă acest lucru nu se acceptă, se poate utiliza procedura de reaşezare "**in situ"** din secvenţa [3.2.10.b].

- În cazul unor aplicaţii particulare, viabilitatea acestei tehnici depinde de lungimea relativă a cheilor şi articolelor.
    - Desigur ea **nu** se justifică pentru articole mici deoarece necesită o zonă de memorie suplimentară pentru tabloul `p` şi timp suplimentar pentru comparaţiile indirecte.
    - Pentru articole de mari dimensiuni se indică de regulă sortarea indirectă, fără a se mai realiza permutarea efectivă a elementelor.
    - Pentru articolele de foarte mari dimensiuni metoda se indică a se utiliza integral, inclusiv permutarea elementelor [Se88].

3.3. Sortarea secvenţelor. Sortarea externă

- Metodele de sortare prezentate în paragraful anterior **nu** pot fi aplicate unor date care **nu** încap în **memoria centrală** a sistemului, dar care pot fi spre exemplu memorate pe dispozitive periferice secvenţiale cum ar fi benzile magnetice.

- În acest caz datele pot fi descrise cu ajutorul unei structuri de tip **secvență** având drept caracteristică esențială faptul că în fiecare moment este accesibilă doar **o singură componentă**.
    - Aceasta este o restricție foarte severă comparativ cu accesul direct oferit de structura tablou, motiv pentru care tehnicile de sortare sunt de cu totul altă natură.
- Una dintre cele mai importante tehnici de sortare a secvențelor este sortarea prin "**interclasare**" (**merging**).

3.3.1. Sortarea prin interclasare

- **Interclasarea** presupune combinarea a două sau mai multe secvențe ordonate într-o singură secvență ordonată, prin selecții repetate ale componentelor curent accesibile.
- Interclasarea este o operație simplă, utilizată ca **auxiliar** în procesul mult mai complex al **sortării secvențiale**.
- O **metodă** de sortare bazată pe interclasare a unei secvențe `a` este următoarea:
    1. Se împarte secvența de sortat `a` în două jumătăți `b` şi `c`.
    2. Se interclasează `b` cu `c`, combinând câte un element din fiecare, în perechi ordonate obținându-se o nouă secvență `a`.
    3. Se repetă cu secvența interclasată `a`, pașii 1 şi 2 de această dată combinând perechile ordonate în qvadruple ordonate.
    4. Se repetă pașii inițiali, interclasând qvadruplele în 8-uple, ş.a.m.d, de fiecare dată dublând lungimea subsecvențelor de interclasare până la sortarea întregii secvențe.
- Spre exemplu fie secvența:

  34 65 12 22 83 18 04 67

    - Execuția pasului 1 conduce la două jumătăți de secvență:

      34 65 12 22
      83 18 04 67

    - Interclasarea componentelor unice în perechi ordonate conduce la secvența

      34 83 | 18 65 | 04 12 | 22 67

    - Înjumătățind din nou şi interclasând perechile în qvadruple se obține:

      04 12 34 83 | 18 22 65 67

    - Cea de-a treia înjumătățire şi interclasarea celor două qvadruple într-un 8-uplu conduc la secvența gata sortată:

      04 12 18 22 34 65 67 83

- Fiecare operație care tratează întregul set de date se numeşte **fază;**
- Procesul prin repetarea căruia se realizează sortarea se numeşte **trecere**.
- Procesul de sortare anterior descris constă din trei treceri fiecare cuprinzând o fază de înjumătățire şi una de interclasare.
- Pentru a realiza sortarea sunt necesare trei secvențe motiv pentru care sortarea se numeşte **interclasare cu trei secvențe**.

- Aceasta este de fapt o **interclasare neechilibrată cu 3 secvențe**.

3.3.2. Interclasarea neechilibrată cu trei secvențe

- Interclasarea neechilibrată cu trei secvențe reprezintă implementarea procedeului de sortare precizat anterior.
- Schema de principiu a acestui procedeu apare în figura 3.3.1.1.

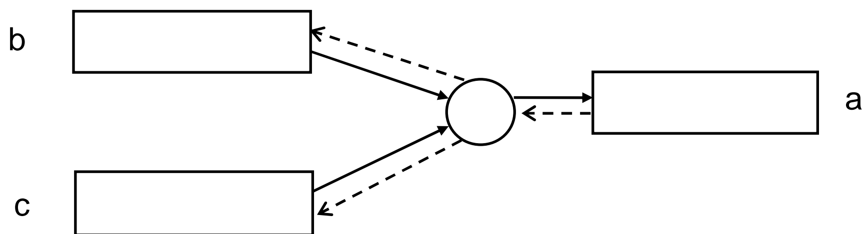


Fig. 3.3.1.1. Interclasare neechilibrată cu trei secvențe

- Într-o primă etapă în secvențele [3.3.1.1.a, b] se prezintă structurile de date şi schița de principiu a algoritmului.
- După cum se observă, fiecare trecere care constă dintr-o reluare a buclei **REPEAT**, conține două faze:
  - O **fază de înjumătățire** adică de distribuție a *n*-uplelor secvenței `a` pe cele două secvențe `b` şi `c`, respectiv
  - O **fază de interclasare** în care *n*-uplele de pe secvențele `b` şi `c` se interclasează în *n*-uple de dimensiune dublă pe secvența `a`.
  - Variabila `p` inițializată pe 1, precizează dimensiunea *n*-uplelor curente, dimensiune care după fiecare trecere se dublează.
    - În consecință numărul total de treceri va fi $\lceil log_2 n \rceil$.

```
------------------------------------------------------------
{Interclasarea neechilibrată cu trei secvențe - Structuri de
date}

TYPE TipElement = RECORD
       cheie: TipCheie;
       {alte campuri}                     [3.3.1.1.a]
     TipSecventa = FILE OF TipElement;
VAR a,b,c: TipSecventa;
------------------------------------------------------------
```

```
{Interclasare neechilibrata cu 3 secvențe - Pas rafinare 0}

PROCEDURE Interclasare3Secvente;
  p:= 1; {dimensiune n-uplu}                        [3.3.1.1.b]
  REPEAT
    *injumatatire; {distribuie a pe b şi c}
    *interclasare; {interclasează de pe b şi c pe a]
    p:= 2*p
  UNTIL k=1; {k este contorul de n-uple}
------------------------------------------------------------
```

- Variabila `k` contorizează numărul de *n*-uple create în procesul de interclasare.

- Procesul de sortare se încheie când în final rămâne un singur *n*-uplu de dimensiune *n* (`k=1`).

- În continuare se procedează la rafinarea celor două faze.

- În secvența [3.3.1.1.c] apare primul pas de rafinare al fazei de înjumătățire, iar în secvența următoare rafinarea enunțului "`scrie un` *n*`-uplu de dimensiune p în secvența d`".

```
------------------------------------------------------------
{Procedura Injumatatire - Pas rafinare 1}

PROCEDURE Injumatatire(p: integer);
{distribuie n-uplele de pe a pe b si c}
{p - dimensiune n-uplu}
  RESET(a); REWRITE(b); REWRITE(c);
  WHILE NOT Eof(a) DO BEGIN                    [3.3.1.1.c]
    *scrie un n-uplu pe b
    *scrie un n-uplu pe c
  END;
------------------------------------------------------------
PROCEDURE ScrieNuplu(d: TipSecventa);
{scrie un n-uplu de dimensiune p în secvența d}
  i:= 0; {contor elemente n-uplu}
WHILE (i<p) AND NOT Eof(a) DO BEGIN
    *citeste(a,x);                             [3.3.1.1.d]
    *scrie(d,x)
    i:= i+1
END;
------------------------------------------------------------
```

- Variabila `i` reprezintă **contorul de elemente** care poate lua valori între 0 şi `p`.

    - Scrierea se **termină** la atingerea numărului `p` de elemente sau la terminarea secvenței sursă.

- Rafinarea fazei de interclasare apare în secvența [3.3.1.1.e].

- Variabila de intrare `p` reprezintă **dimensiunea** *n*-uplelor care se interclasează, iar `k` este **contorul** de *n*-uple.

- Practic interclasarea propriu-zisă (bucla `REPEAT`) se încheie la terminarea prelucrării secvențelor `b` şi `c`.

```
------------------------------------------------------------
{Procedura Intercalsare - Pas rafinare 1}

PROCEDURE Interclasare(p: integer; VAR k: integer);
  {p - dimensiune n-uplu, k - contor n-uple}
Rewrite(a); Reset(b); Reset(c);
k:= 0;                                        [3.3.1.1.e]
*iniţializare interclasare
*citeste în x respectiv în y primul element din b
   respectiv din c {tehnica lookahead}
REPEAT
  *interclaseaza câte un n-uplu de pe b si c pe a
UNTIL EndPrelucr_b AND EndPrelucr_c
Close(a); Close(b); Close(c);
------------------------------------------------------------
```

- Datorită particularităţilor de implementare a fişierelor sunt necesare câteva precizări:

    - Variabila `Eof(f)` se poziţionează pe **`true`** la citirea ultimului element al fişierului f.

    - Citirea dintr-un fişier cu `Eof` poziţionat pe **`true`** conduce la **eroare**.

    - Din punctul de vedere al algoritmului de interclasare, terminarea prelucrării unui fişier **nu** coincide cu poziţionarea lui `Eof` pe **`true`**, deoarece mai trebuie prelucrat ultimul element citit.

- Pentru rezolvarea acestor constrângeri se utilizează **tehnica scrutării** (lookahead).

    - Tehnica scrutării constă în introducerea unei **întârzieri** între momentul citirii şi momentul prelucrării unui element.

    - Astfel în fiecare moment se prelucrează elementul citit în **pasul anterior** şi se citeşte un **nou element**.

    - În acest scop pentru fiecare fişier implicat în prelucrare se utilizează:

        - (1) O variabilă specială de `TipElement` care memorează elementului curent

        - (2) O variabilă booleană `EndPrelucrare` a cărei valoare **`true`** semnifică terminarea prelucrării ultimului element al fişierului.

- Rafinarea enunţului "`interclasează câte un` *n*`-uplu de pe b şi c pe a`" apare în secvenţa [3.3.1.1.f] care aplică tehnica anterior precizată.

- Variabilele specifice asociate secvenţelor `b` şi `c` sunt `x` şi `y` respectiv `EndPrelucr_b` şi `EndPrelucr_c`.

```
------------------------------------------------------------
  {interclaseaza câte un n-uplu de pe b si c pe a}
  i:= 0; {contor n-uplu b}
  j:= 0; {contor n-uplu c}
  WHILE (i<p)AND(j<p) AND NOT EndPrelucr_b AND
             NOT EndPrelucr_c DO
    BEGIN
      IF x.cheie<y.cheie THEN BEGIN
```

```
           *scrie(a,x); i:= i+1;
           *citeste(b,x)                       [3.3.1.1.f]
          END
        ELSE BEGIN
           *scrie(a,y); j:= j+1;
           *citeste(c,y)
          END
  END; {WHILE}
  *copiază restul n-uplului de pe b pe a (dacă există)
  *copiază restul n-uplului de pe c pe a (dacă există)
  k:= k+1;
------------------------------------------------------------
```

- O variantă de implementare integrală a procesului de sortare neechilibrată cu 3 benzi apare în **PROCEDURA** `Interclasare3Secvente` secvența [3.3.1.1.g].

```
------------------------------------------------------------
{Procedura Interclasare 3 secvențe}

PROCEDURE Interclasare3Secvente;
    VAR a,b,c: TipSecventa;
        p,k: integer;

    PROCEDURE Injumatatire(p: Integer);
      VAR x: TipElement;
      PROCEDURE ScrieNuplu(VAR d: TipBanda);
        VAR i: integer;
        BEGIN {ScrieNuplu}
          i:= 0;
          WHILE (i<n) AND (NOT Eof(a)) DO BEGIN
            Read(a,x);
            Write(d,x); i:= i+1
          END; {WHILE}
      END; {ScrieNuplu}                            [3.3.1.1.g]
      BEGIN {Injumatatire}
        Reset(a); Rewrite(b); Rewrite(c);
        WHILE NOT Eof(a) DO BEGIN
          ScrieNuplu(b); ScrieNuplu(c);
        END; {WHILE}
        Close(a); Close(b); Close(c);
    END; {Injumatatire}

    PROCEDURE Interclasare(p: integer; VAR k: integer);
      VAR i,j: integer;
          x,y: TipElement;
          EndPrelucr_b,EndPrelucr_c: Boolean;
      BEGIN {Interclasare}
        Reset(b); Reset(c); Rewrite(a); k:= 0;
        EndPrelucr_b:= Eof(b); EndPrelucr_c:= Eof(c);
        IF NOT EndPrelucr_b THEN Read(b,x); {lookahead}
        IF NOT EndPrelucr_c THEN Read(c,y); {lookahead}
        REPEAT
          i:= 0; j:= 0;     {interclasarea unui n-uplu}
          WHILE (i<p)AND(j<p) AND NOT EndPrelucr_b AND
                  NOT EndPrelucr_c DO BEGIN
            IF x.cheie < y.cheie THEN
                BEGIN
                  Write(a,x); i:= i+1;
                  IF Eof(b) THEN EndPrelucr_b:= true
```

```
                    ELSE
                      Read(b,x)
                END
              ELSE
                BEGIN
                  Write(a,y); j:= j+1;
                  IF Eof(c) THEN EndPrelucr_c:= true
                    ELSE
                      Read(c,y)
                END;
        END; {WHILE}
        {copiază restului n-uplului de pe b pe a}
        WHILE (i<n) AND NOT EndPrelucr_b DO BEGIN
          Write(a,x); i:= i+1;
          IF Eof(b) THEN
              EndPrelucr_b:= true
            ELSE
              Read(b,x)
        END; {WHILE}
        {copiază restului n-uplului de pe c pe a}
        WHILE (j<n) AND NOT EndPrelucr_c DO BEGIN
          Write(a,y); j:= j+1;
          IF Eof(c) THEN
              EndPrelucr_c:= true
            ELSE
              Read(c,y)
        END; {WHILE}
        k:= k+1;
      UNTIL EndPrelucr_b AND EndPrelucr_c;
      Close(a); Close(b); Close(c);
    END; {Interclasare}
    BEGIN {Interclasare3Secvente}
      p:= 1;
      REPEAT
        Injumatatire(p);                  {faza (1)}
        Interclasare(p,k);                {faza (2)}
        p:= p*2;
      UNTIL k=1;
END; {Interclasare3Secvente}
------------------------------------------------------------
```

3.3.3. Principiul sortării prin interclasare naturală

- Tehnica de sortare prin interclasare **nu** ia în considerare faptul că datele inițiale pot fi parțial sortate, subsecvențele având o lungime predeterminată (2^k în trecerea k).
- De fapt, oricare două subsecvențe ordonate de lungimi m și n pot fi interclasate într-o singură subsecvență ordonată de lungime m+n.
- Tehnica de interclasare care în fiecare moment combină cele mai lungi secvențe ordonate posibile se numește **sortare prin interclasare naturală.**
- În cadrul acestei tehnici un rol central îl joacă noțiunea de **monotonie**, care va fi clarificată pe baza următorului exemplu.
- Se consideră următoarea secvență de chei,

|1 13 | 2 4 7 | 6 18 | 9 10 14 | 11 | 3 75 |

    - Se pun linii verticale la extremitățile secvenței precum şi între elementele a_j şi a_{j+1}, ori de câte ori $a_j > a_{j+1}$.

    - În felul acesta secvența a fost defalcată în **secvențe parțiale monotone**.

    - Acestea sunt de **lungime maximă**, adică **nu** pot fi prelungite fără a-şi pierde proprietatea de a fi monotone.

- În general fie $a_1, a_2, ..., a_n$ o secvență oarecare de numere întregi. Se înțelege prin **monotonie** orice secvență parțială $a_i, ..., a_j$ care satisface următoarele condiții:

  1) $1 \le i \le j \le n$;

  2) $a_k \le a_{k+1}$ pentru orice $i \le k < j$;

  3) $a_{i-1} > a_i$ sau $i = 1$; [3.3.2.a]

  4) $a_j > a_{j+1}$ sau $j = n$.

- Această definiție include şi monotoniile cu un singur element, deoarece în acest caz `i=j` şi proprietatea 2 este îndeplinită, neexistând nici un `k` cuprins între `i` şi `j-1`.

- Sortarea naturală interclasează **monotonii**.

- Sortarea se bazează pe următoarea **proprietate**:

    - Dacă se intercalează două secvențe a câte `n` monotonii fiecare, rezultă o secvență cu exact `n` monotonii.

    - Ca atare la fiecare trecere numărul acestora se înjumătățeşte şi în cel mai rău caz, numărul necesar de **mişcări** este $n*\lceil \log_2 n \rceil$, în medie mai redus.

    - Numărul de **comparații** este însă mult mai mare deoarece pe lângă comparațiile necesare interclasării elementelor sunt necesare comparații între elementele consecutive ale fiecărui fişier pentru a determina sfârşitul fiecărei monotonii.

- În continuare în dezvoltarea programului aferent acestei tehnici va fi utilizată **metoda detalierilor succesive**.

- Se va utiliza o structură de date de tip **fişier secvențial** asupra căreia se va aplica sortarea prin interclasare **neechilibrată** în două faze, utilizând trei secvențe.

- Algoritmul lucrează cu secvențele `a, b` şi `c`. Secvența `c` este cea care trebuie procesată şi care în final devine secvența sortată. În practică, din motive de securitate, `c` este de fapt o copie a secvenței inițiale.

- Se utilizează următoarele structuri de date [3.3.2.b].

```
--------------------------------------------------------
TYPE  TipSecventa = FILE OF TipElement;        [3.3.2.b]
VAR a,b,c: TipSecventa;
--------------------------------------------------------
```

- Secvențele `a` şi `b` sunt auxiliare şi ele servesc la defalcarea provizorie a lui `c` pe monotonii.
- Fiecare trecere constă din două faze alternative care se numesc **defalcare** respectiv **interclasare**.
    - În faza de defalcare monotoniile secvenței `c` se defalcă alternativ pe secvențele `a` şi `b`.
    - În faza de interclasare se recombină în `c`, monotoniile de pe secvențele `a` şi `b` (fig. 3.3.2).

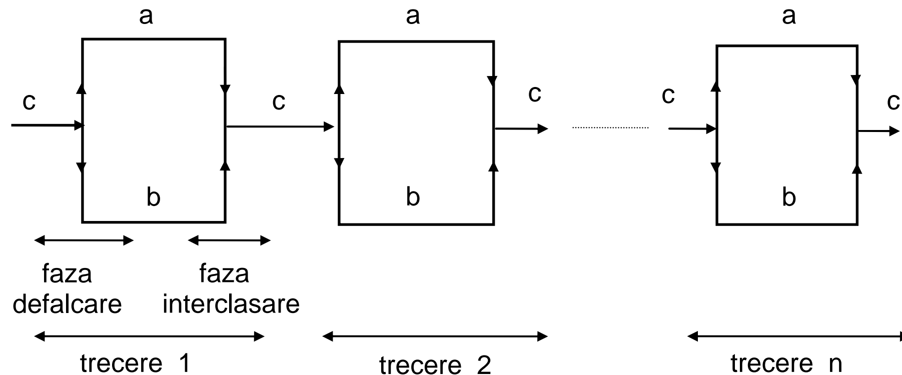


Fig. 3.3.2. Treceri şi faze în interclasarea naturală

- Sortarea se termină în momentul în care numărul monotoniilor secvenței `c` devine egal cu 1.
- Pentru numărarea monotoniilor se utilizează variabila `l`.
- Prima formă a algoritmului apare în secvența [3.3.2.c].
    - Cele două faze apar ca două instrucții, care în continuare urmează să fie rafinate.
    - Procesul de rafinare poate fi realizat
        - Fie prin substituția directă a celor două instrucții cu secvențele care le corespund (**tehnica inserției**),
        - Fie prin interpretarea lor ca proceduri şi procedând în consecință la dezvoltarea lor (**tehnica selecției**).

```
------------------------------------------------------------
PROCEDURE InterclasareNaturala;
VAR l: integer;
    a,b,c: TipSecventa;
    sm: boolean;
BEGIN                                          [3.3.2.c]
  REPEAT
    Rewrite(a); Rewrite(b); Reset(c);
    Defalcare;
    Reset(a); Reset(b); Rewrite(c);
    l:= 0;
    Interclasare;
  UNTIL l=1
END; {InterclasareNaturala}
--------------------------------------------------------
```

* În continuare se va continua procesul de rafinare prin tehnica **selecției**.

* În secvențele [3.3.2.d] respectiv [3.3.2.e] apar primii paşi de rafinare pentru `Defalcare` respectiv `Interclasare`.

```
------------------------------------------------------------
PROCEDURE Defalcare; {din c pe a si b}
BEGIN
  REPEAT                                              [3.3.2.d]
    CopiazaMonotonia(c,a);
    IF NOT Eof(c) THEN CopiazaMonotonia(c,b)
  UNTIL Eof(c)
END; {Defalcare}
------------------------------------------------------------
```

* Această metodă de defalcare distribuie fie un număr egal de monotonii pe secvențele `a` respectiv `b`, fie cu o monotonie mai mult pe secvența `a`, după cum numărul de monotonii de pe secvența `c` este par respectiv impar.

```
------------------------------------------------------------
PROCEDURE Interclasare;
  BEGIN {din a si b pe c}
    REPEAT
      InterclasareMonotonie; l:= l+1;
    UNTIL Eof(b);
    IF NOT Eof(a) THEN                                [3.3.2.e]
      BEGIN {monotonia nepereche}
        CopiazaMonotonia(a,c); l:= l+1
      END
  END; {Interclasare};
------------------------------------------------------------
```

* După interclasarea monotoniilor perechi, monotonia nepereche (dacă există) trebuie recopiată pe `c`.

* Procedurile `Defalcare` şi `Interclasare` sunt redactate în termenii unor proceduri subordonate (`InterclasareMonotonie`, `CopiazaMonotonia`) care se referă la o singură monotonie şi care vor fi rafinate în continuare în [3.3.2.f] respectiv [3.3.2.g].

* Se introduce variabila booleană `sm` (sfârşit monotonie) care specifică dacă s-a ajuns sau **nu** la sfârşitul unei monotonii. La epuizarea uneia dintre monotonii restul celeilalte este copiat în secvența destinație.

```
------------------------------------------------------------
PROCEDURE CopiazaMonotonia( x,y: TipSecventa);
   {x - fişierul in care se delimitează monotonia
    y - fişierul in care se copiază monotonia}
  BEGIN
    REPEAT                                            [3.3.2.f]
      CopiazaElement(x,y)
    UNTIL sm
  END; {CopiazaMonotonia}
------------------------------------------------------------
PROCEDURE InterclasareMonotonie;
  BEGIN
    REPEAT
      IF a.elemCurent.cheie < b.elemCurent.cheie THEN
          BEGIN
            CopiazaElement(a,c);
```

```
          IF sm THEN CopiazaMonotonia(b,c)
        END                                   [3.3.2.g]
      ELSE
        BEGIN
          CopiazaElement(b,c);
           IF sm THEN CopiazaMonotonia(a,c)
         END {ELSE}
   UNTIL sm
 END; {InterclasareMonotonie}
------------------------------------------------------------
```

- Pentru redactarea procedurilor de mai sus se utilizează o procedură subordonată `CopiazaElement(x,y: TipSecventa)`, care transferă elementul curent al secvenței sursă `x` în secvența destinație `y`, poziționând variabila `sm` funcție de atingerea sau **nu** a sfârşitului monotoniei.

- În acest scop se utilizează **tehnica "lookahead"** (scrutare în față), bazată pe citirea în pasul curent a elementului pentru pasul următor, primul element fiind introdus în tamponul fişierului înaintea demarării procesului de defalcare respectiv de interclasare.

- Pentru acest scop se modifică şi structura de date aferentă secvenței după cum urmează [3.3.2.h].

```
------------------------------------------------------------
TYPE TipSecventa = RECORD                     [3.3.2.h]
       secventa: FILE  OF  TipElement;
       elemCurent: TipElement; {tamponul fisierului}
       termPrelucr: boolean
     END;
------------------------------------------------------------
```

Procedura `CopiazaElement` apare în secvența [3.3.2.i].

```
------------------------------------------------------------
PROCEDURE CopiazaElement(VAR x,y: TipSecventa);
  VAR aux: TipElement;
  BEGIN
    Write(y.secvența,x.elemCurent);
    IF Eof(x.secventa) THEN
        BEGIN
          sm:= true; x.termPrelucr:= true
        END                                   [3.3.2.i]
      ELSE
        BEGIN
          aux:= x.elemCurent;
          Read(x.secventa,x.elemCurent);
          sm:= aux.cheie > x.elemCurent.cheie
        END;
  END; {CopiazaElement}
------------------------------------------------------------
```

- După cum se observă:

  - La momentul **curent** se scrie pe secvența destinație `y` elementul `x.elemCurent` citit în pasul anterior

  - Se citeste noul `x.elemCurent` în vederea determinării sfârşitului de monotonie `sm` sau a sfârşitului prelucrării `termPrelucr`. În acest scop se utilizează variabila `aux: TipElement`.

* Desigur unii dintre paşii de rafinare precizaţi pot suferi anumite modificări, funcţie de natura secvenţelor reale care se utilizează şi de setul de operaţii disponibile asupra acestora.

* Din păcate, programul dezvoltat cu ajutorul acestei metode **nu** este corect în toate cazurile.

  * Spre exemplu, defalcarea secvenţei `c` cu 10 monotonii:

    |13 57|17 19|11 59|23 29|7 61|31 37|5 67|41 43|2 3|47 71|

    are drept consecinţă datorită distribuţiei cheilor formarea a 5 monotonii pe secvenţa `a` şi a unei singure monotonii pe secvenţa `b`, în loc de 5 cum era de aşteptat.

    `a`: |13 57|11 59|7 61|5 67|2 3|

    `b`: |17 19 23 29 31 37 41 43 47 71|

  * Faza de interclasare conduce la o secvenţă cu două monotonii (în loc de 5)

    `c`: |13 17 19 23 29 31 37 41 43 47 57 71|11 59|

    deoarece în procesul de interclasare s-a ajuns la sfârşitul secvenţei `b` şi conform lui [3.3.2.d] se mai copiază o **singură** monotonie din `a`. După trecerea următoare sortarea se încheie, dar rezultatul este incorect:

    `c`:|11 13 17 19 23 29 31 37 41 43 47 57 59 71|

* Acest lucru se întâmplă deoarece **nu** a fost luat în considerare faptul că deşi procesul de distribuire repartizează în mod egal monotoniile pe secvenţele `a` respectiv `b`, numărul de monotonii pe cele două secvenţe poate **diferi** foarte mult datorită distribuţiei cheilor.

* Pentru a remedia această situaţie, este necesar ca procedura `Interclasare` să fie **modificată** astfel încât, în momentul în care se ajunge la sfârşitul unei secvenţe, să copieze în `c` **tot** restul celeilalte secvenţe.

* Versiunea revizuită a algoritmului de sortare prin interclasare naturală apare în [3.3.2.j].

```
------------------------------------------------------------
PROCEDURE InterclasareNaturala;
  VAR l: integer;
      sm: boolean;
      a,b,c: TipSecventa;
  PROCEDURE CopiazaElement(VAR x,y: TipSecventa);
    VAR aux: TipElement;
    BEGIN
      Write(y.secventa,x.elemCurent);
      IF Eof(x.secventa) THEN
          BEGIN
            sm:= true;
            x.termPrelucr:= true
          END                                   [3.3.2.j]
        ELSE
          BEGIN
            aux:= x.elemCurent;
```

```
          Read(x.secventa,x.elemCurent);
          sm:= aux.cheie > x.elemCurent.cheie
        END;
  END; {CopiazaElement}
PROCEDURE CopiazaMonotonia(VAR x,y: TipSecventa);
  BEGIN
    REPEAT
      CopiazaElement(x,y)
    UNTIL sm
  END; {CopiazaMonotonia}
PROCEDURE Defalcare;
  BEGIN
    Rewrite(a.secventa);Rewrite(b.secventa);
    Reset(c.secventa);
    c.termPrelucr:= Eof(c.secventa);
    Read(c.secventa,c.elemCurent);
    REPEAT
      CopiazaMonotonia(c,a);
      IF NOT c.termPrelucr THEN
        CopiazaMonotonia(c,b)
    UNTIL c.termPrelucr;
    Close(a.secventa); Close(b.secventa);
    Close(c.secventa)
  END; {Defalcare}
PROCEDURE InterclasareMonotonie;
  BEGIN
    REPEAT
      IF a.elemCurent.cheie < b.elemCurent.cheie THEN
          BEGIN
            CopiazaElement(a,c);
            IF sm THEN CopiazaMonotonia(b,c)
          END
        ELSE
          BEGIN
            CopiazaElement(b,c);
            IF sm THEN CopiazaMonotonia(a,c)
          END
    UNTIL sm                      [3.3.2.j] continuare
  END; {InterclasareMonotonie}
PROCEDURE Interclasare;
  BEGIN
    Reset(a.secventa); Reset(b.secventa);
    Rewrite(c.secventa);
    a.termPrelucr:= Eof(a.secventa);
    b.termPrelucr:= Eof(b.secventa);
    IF NOT a.termPrelucr THEN
      Read(a.secventa,a.elemCurent); {primul element}
    IF NOT b.termPrelucr THEN
      Read(b.secventa,b.elemCurent); {primul element}
    WHILE NOT b.termPrelucr DO
      BEGIN
        InterclasareMonotonie; l:= l+1
      END; {WHILE}
    IF NOT a.termPrelucr THEN
      BEGIN
        CopiazaMonotonia(a,c); l:= l+1
      END; {IF}
    Close(a.secventa);Close(b.secventa);
```

```
        Close(c.secventa);
      END; {Interclasare
    BEGIN {InterclasareNaturala}
      REPEAT
        Defalcare;
        l:= 0;
        Interclasare;
      UNTIL l=1;
END; {InterclasareNaturala};
------------------------------------------------------------
```

- **Analiza sortării prin interclasare naturală.**

- După cum s-a mai precizat, la analiza unei metode de sortare externă, numărul comparaţiilor de chei **nu** are importanţă practică, deoarece durata prelucrărilor în unitatea centrală a sistemului de calcul este neglijabilă faţă de durata acceselor la memoriile externe.

- Din acest motiv **numărul mutărilor M** va fi considerat drept **unic** indicator de performanţă.

- În cazul sortării prin intercalsare naturală:

  - La o trecere, în fiecare din cele două faze (defalcare şi interclasare) se mută toate elementele, deci $M = 2*n$.

  - După fiecare trecere numărul monotoniilor se micşorează de două ori, uneori chiar mai substanţial, motiv pentru care a fost necesară şi modificarea anterioară a procedurii `Interclasare`.

  - Ştiind că numărul maxim de monotonii iniţiale este `n`, numărul maxim de **treceri** este $\lceil \log_2 n \rceil$, astfel încât în cel mai defavorabil caz numărul de **mişcări** $M=2*n*\lceil \log_2 n \rceil$, în medie simţitor mai redus.

3.4. Rezumat

- **Sortarea** este domeniul ideal al studiului atât al construcţiei algoritmilor cât şi al tehnicilor de programare aferente.
- Prin **sortare** se înţelege în general **ordonarea** unei mulţimi de elemente, cu scopul de a facilita căutarea ulterioară a unui element dat.
- Algoritmii de sortare se clasifică în două mari categorii: **sortarea tablourilor** numită **sortare internă** şi **sortarea fişierelor** (secvenţelor) numită şi **sortare externă.**
- Metodele cele mai cunoscute de sortare **"in situ"** a tablourilor **sunt sortarea prin inserţie, sortarea prin selecţie** şi **sortarea prin interschimbare**. Acestea se numesc **metode directe de sortare** şi au în general performanţe de ordinul $O(n^2)$
- Dintre **metodele avasnsate de sortare** cele mai cuneoscute sunt **sortarea prin metoda ansamblelor** şi **sortarea prin partiţionare (quicksort)**. Aceste metode au performanţe de ordinul $O(n \log_2 n)$.
- Prin generalizare metoda partiţionării poate determina cu performaţa $O(n)$**, cel de-al k-lea element al oricărui tablou de elemente**, indiferent de valoarea lui k.
- Dacă cunoaştem cu precizie **domeniul cheilor unui tablou** şi renunţăm la constrângerea **in situ**, putem accelera procesul de sortare. Un exemplu îl reprezintă **tehnica binsort** care poate fi implementată prin **determinarea distribuţiei cheilor**.
- Dacă tablourile de sortat au **elemente de mari dimensiuni** se poate utiliza **tehnica sortării indirecte.**

- **Sortarea secvenţelor** sau **sortarea externă** se realizează după alte principii de sortare respectiv prin **interclasare**.
- Se cunosc mai multe tipuri de interclasări dintre care se **amintesc interclasarea neechilibrată cu trei secvenţe, interclasarea echilibrată cu 4 secvenţe** şi **interclasarea naturală**. Ele presupun implementări complet diferite de cele specifice sortării interne.

3.5. Exerciţii

1) Ce este *sortarea*? Ce fel de *tipuri de sortări* cunoaşteţi?
2) Care sunt cele mai cunoscute *metode de sortare directe* ale tablourilor? Descrieţi principial fiecare dintre aceste sortări.
3) Se cere să se redacteze câte o funcţie pentru fiecare din *tipurile de sortări directe*. Funcţiile nu vor returna nici un rezultat. Fiecare funcţie primeşte ca parametru tabloul de sortat şi dimensiunea acestuia.
4) Se cere să se redacteze un *program C* care:
   a) defineşte tablourile de numere întregi a, b şi c
   b) citeşte elementele tablourilor de la tastatură
   c) sortează tabloul a prin interschimbare, tabloul b prin selecţie şi tabloul c prin interschimbare utilizând funcţiile de la aplicaţia anterioară
   d) afişează tablourile sortate
5) Ce este un *ansamblu*? Cum se realizează *deplasarea de sus în jos* într-un *ansamblu*? Cum se poate *implementa un ansamblu*? Precizaţi la nivel pseudocod principiul *deplasării de sus în jos* într-un *ansamblu*.
6) Care este principiul *sortării prin metoda ansamblelor*? Estimaţi performanţa acestei metode de sortare. Implementaţi în limbajul C sortarea prim metoda ansamblelor. Se va utiliza structura de date ansamblu şi operatorul deplasare definit pe aceasta.
7) Ce este *partiţionarea*? Care este *principiul sortării prin partiţionare*? Redactaţi o funcţie C care implementează sortarea prin partiţionare a unui tablou de dimensiune precizată. Care este performanţa acestei metode de sortare?
8) Care este principiul *determinării prin partiţionare a medianei unui tablou*? Redactaţi funcţia C care găseşte cel de-al k-lea element al unui tablou dat. Care este performanţa acestei metode?
9) Care este *principiul metodei binsort*? Care sunt premisele utilizării acestei metode?
10) Se cere să se redacteze funcţia C ca realizează implementarea sortării cu determinarea distribuţiilor cheilor pentru m=10. Fiind dat tabloul de întregi 0,1,3,5,8,4,3,2,1,4,8,7,5,6,9 se cere să se explice paşii procesului de sortare a acestui tablou utilizînd sortarea cu determinarea distribuţiei cheilor.
11) Ce este *sortarea indirectă* şi unde se aplică ea? Scrieţi o funcţie C care implementează sortarea indirectă pornind de las sortarea prin selecţie.
12) Ce este *interclasarea*? Explicaţi principiul sortării prin *interclasare neechilibrată cu trei secvenţe*. Exemplificaţi pe secvenţa 17, 56, 11, 27, 76, 19, 7, 62. Care este performaţa acestei metode de sortare?
13) Ce este *principiul sortării prin interclasare naturală*? Care este *modelul* acestei sortări? Care este performaţa acestei metode de sortare?