Capitolul 4

Analiza algoritmilor de sortare

4.1 Problematica sortării

Se consideră un set finit de obiecte, fiecare având asociată o caracteristică, numită *cheie*, ce ia valori într-o mulţime pe care este definită o relaţie de ordine. *Sortarea* este procesul prin care elementele setului sunt rearanjate astfel încât cheile lor să se afle într-o anumită ordine.

Exemplul 4.1 Considerăm setul de valori întregi: (5,8,3,1,6). În acest caz cheia de sortare coincide cu valoarea elementului. Prin sortare *crescătoare* se obţine setul (1,3,5,6,8) iar prin sortare *descrescătoare* se obţine (8,6,5,3,1).

Exemplul 4.2 Considerăm un tabel constând din nume ale studenţilor şi note: ((Popescu,9), (Ionescu,10), (Voinescu,8), (Adam,9)). În acest caz cheia de sortare poate fi numele sau nota. Prin ordonare crescătoare după nume se obţine ((Adam,9), (Ionescu,10), (Popescu,9), (Voinescu,8)), iar prin ordonare descrescătoare după notă se obţine ((Ionescu,10), (Popescu,9), (Adam,9), (Voinescu,8)).

Pentru a simplifica prezentarea, în continuare vom considera că setul prelucrat este constituit din valori scalare ce reprezintă chiar cheile de sortare şi că scopul urmărit este ordonarea crescătoare a acestora. Astfel, a ordona setul $(x_1, x_2, \ldots, x_n)$ este echivalent cu a găsi o permutare de ordin n, $(p(1), p(2), \ldots, p(n))$ astfel încât $x_{p(1)} \leq x_{p(2)} \leq \ldots \leq x_{p(n)}$.

De asemenea vom considera că elementele setului sunt stocate pe un suport de informaţie ce permite accesul aleator la date. În acest caz este vorba despre *sortare internă*. În cazul în care suportul de informaţie permite doar accesul secvenţial la date trebuie folosite metode specifice încadrate în categoria metodelor de *sortare externă*.

Pe de altă parte, în funcţie de spaţiul de manevră necesar pentru efectuarea sortării există:

- Sortare ce foloseşte o zonă de manevră de dimensiunea setului de date. Dacă setul iniţial de date este reprezentat de tabloul $x[1..n]$, cel sortat se va obţine într-un alt tablou $y[1..n]$.
- Sortare în aceeaşi zonă de memorie (sortare pe loc - *in situ*). Elementele tabloului $x[1..n]$ îşi schimbă poziţiile astfel încât după încheierea procesului să fie ordonate. Este posibil ca şi în acest caz să se folosească o zonă de memorie însă aceasta este de regulă de dimensiunea unui element şi nu de dimensiunea întregului tablou.

În continuare vom analiza doar metode de sortare internă în aceeaşi zonă de memorie. Metodele de sortare pot fi caracterizate prin:

Stabilitate. O metodă de sortare este considerată stabilă dacă ordinea relativă a elementelor ce au aceeaşi valoare a cheii nu se modifică în procesul de sortare. De exemplu dacă asupra tabelului cu note din Exemplul 4.2 se aplică o metodă stabilă de ordonare descrescătoare după notă se obţine ((Ionescu,10), (Popescu,9), (Adam,9), (Voinescu,8)) pe când dacă se aplică una care nu este stabilă se va putea obţine ((Ionescu,10), (Adam,9), (Popescu,9), (Voinescu,8)).

Naturaleţe. O metodă de sortare este considerată naturală dacă numărul de operaţii scade odată cu *distanţa* dintre tabloul iniţial şi cel sortat. O măsură a acestei distanţe poate fi numărul de inversiuni al permutării corespunzătoare tabloului iniţial.

Eficienţă. O metodă este considerată eficientă dacă nu necesită un volum mare de resurse. Din punctul de vedere al spaţiului de memorie o metodă de sortare pe loc este mai eficientă decât una bazată pe o zonă de manevră de dimensiunea tabloului. Din punct de vedere al timpului de execuţie este important să fie efectuate cât mai puţine operaţii. În general, în analiză se iau în considerare doar operaţiile efectuate asupra elementelor tabloului (comparaţii şi mutări). O metodă este considerată *optimală* dacă ordinul său de complexitate este cel mai mic din clasa de metode din care face parte.

Simplitate. O metodă este considerată simplă dacă este intuitivă şi uşor de înţeles.

Primele două proprietăţi sunt specifice algoritmilor de sortare pe când ultimele sunt cu caracter general. În continuare vom considera câteva metode *elementare* de sortare caracterizate prin faptul că sunt simple, nu sunt cele mai eficiente metode, dar reprezintă punct de pornire pentru metode avansate. Pentru fiecare dintre aceste metode se va prezenta: principiul, verificarea corectitudinii şi analiza complexităţii.

4.2 Sortare prin inserţie

4.2.1 Principiu

Ideea de bază a acestei metode este:

> *Începând cu al doilea element al tabloului $x[1..n]$, fiecare element este inserat pe poziţia adecvată în subtabloul care îl precede.*

La fiecare etapă a sortării, elementului $x[i]$ i se caută poziţia adecvată în subtabloul destinaţie $x[1..i-1]$ (care este deja ordonat) comparând pe $x[i]$ cu elementele din $x[1..i-1]$ începând cu $x[i-1]$ şi creând spaţiu prin deplasarea spre dreapta a elementelor mai mari decât $x[i]$. Astfel structura generală este descrisă în algoritmul `inserţie`.

`inserţie`(**real** $x[1..n]$)
for $i \leftarrow 2, n$ **do**
  $\langle$ inserează $x[i]$ în subşirul $x[1..i-1]$ astfel încât $x[1..i]$ să fie ordonat$\rangle$
end for

Sortarea prin inserţie este detaliată în Algoritmul 4.1.

Algoritmul 4.1 Sortare prin inserţie

```
inserţie1(real x[1..n])
for i ← 2, n do
  j ← i − 1
  aux ← x[i]
  while j ≥ 1 and aux < x[j] do
    x[j + 1] ← x[j]
    j ← j − 1
  end while
  x[j + 1] ← aux
end for
return x[1..n]
```

Există şi alte metode de implementare a metodei. Astfel, pentru a evita efectuarea comparaţiei $j \geq 1$ la fiecare iteraţie interioară se plasează pe poziţia 0 în tabloul x valoarea lui $x[i]$, această poziţie jucând rolul unui *fanion*. Astfel cel mai târziu când $j = 0$ se ajunge la $x[j] = x[i]$. Această variantă este descrisă în Algoritmul 4.2.

4.2.2 Verificarea corectitudinii

Precondiţia problemei este $P = \{n \geq 1\}$, iar postcondiţia este $Q = \{x[1..n]$ este ordonat crescător$\}$. Pentru ciclul exterior (după i) demonstrăm că proprietatea invariantă este $\{x[1..i-1]$ este ordonat crescător$\}$. La începutul ciclului $i = 2$,

deci $x[1..1]$ poate fi considerat crescător. La sfârşitul prelucrării ($i = n + 1$) invariantul implică evident postcondiţia. Rămâne să arătăm că proprietatea rămâne adevărată şi după efectuarea prelucrărilor din cadrul ciclului. Pentru aceasta este suficient să arătăm că pentru ciclul interior (după j) aserţiunea $\{x[1..j]$ este crescător şi $aux \leq x[j+1] = x[j+2] \leq \ldots \leq x[i]\}$ este invariantă. La sfârşitul ciclului **while** această proprietate ar implica una dintre relaţiile:

- $x[1] \leq \ldots \leq x[j] \leq aux \leq x[j+1] = x[j+2] \leq \ldots \leq x[i]$ în cazul în care condiţia de ieşire din ciclu este $aux \geq x[j]$;
- $aux \leq x[1] = x[2] \leq \ldots \leq x[i]$ în cazul în care condiţia de ieşire din ciclu este $j = 0$.

Oricare dintre aceste două relaţii ar conduce prin atribuirea $x[j+1] \leftarrow aux$ la $x[1] \leq \ldots \leq x[j] \leq x[j+1] \leq x[j+2] \leq \ldots \leq x[i]$ iar prin trecerea la următoarea valoare a contorului ($i \leftarrow i + 1$) la faptul că $x[1..i-1]$ este crescător.

Rămâne doar să justificăm invariantul ciclului **while**. La intrarea în ciclu au loc relaţiile: $j = i - 1$, $aux = x[i]$ deci $aux = x[j+1] = x[i]$ iar cum $x[1..i-1]$ este crescător rezultă că proprietatea invariantă propusă pentru **while** este satisfăcută. Arătăm că ea nu este alterată de prelucrările din cadrul ciclului: dacă $aux < x[j]$, prin atribuirea $x[j+1] \leftarrow x[j]$ se obţine: $aux < x[j] = x[j+1] \leq \ldots \leq x[i]$ iar după $j \leftarrow j - 1$ va fi adevărată aserţiunea: $\{$ $x[1..j]$ crescător şi $aux < x[j+1] = x[j+2] \leq \ldots \leq x[i]$. Cum $x[j+1] = x[j+2]$ prin atribuirea $x[j+1] \leftarrow x[j]$ nu se pierde informaţie din tablou.

Oprirea algoritmului este asigurată de utilizarea câte unui contor pentru fiecare dintre cele două cicluri.

Algoritmul 4.2 Sortarea prin inserţie cu folosirea unui fanion

```
inserţie2(real x[1..n])
for i ← 2, n do
   x[0] ← x[i]
   j ← i − 1
   while x[0] < x[j] do
      x[j + 1] ← x[j]
      j ← j − 1
   end while
   x[j + 1] ← x[0]
end for
return x[1..n]
```

4.2.3 Analiza complexităţii

Vom lua în considerare doar operaţiile de comparare şi mutare (atribuire) efectuate asupra elementelor tabloului. Fie $T_C(i)$ şi $T_M(i)$ numărul comparaţiilor

respectiv al atribuirilor care implică elementele tabloului pentru fiecare $i = \overline{2, n}$. In funcţie de problema concretă de rezolvat poate fi mai mare costul comparaţiilor (dacă compararea presupune efectuarea mai multor prelucrări) sau costul atribuirilor (dacă elementele tabloului sunt structurate).

Cazul cel mai favorabil corespunde şirului ordonat crescător iar cel mai defavorabil celui ordonat descrescător.

În cazul cel mai favorabil $T_C(i) = 1$, iar $T_M(i) = 2$ (este vorba de operaţiile care implică variabila ajutătoare aux şi care dealtfel în acest caz nu au nici un efect) deci $T(n) \geq \sum_{i=2}^{n}(T_C(i)+T_M(i)) = 3(n-1)$. În cazul cel mai defavorabil $T_C(i) = i$ iar $T_M(i) = i-1+2 = i+1$, obţinându-se că $T(n) \leq \sum_{i=2}^{n}(2i+1) = n^2 + 2n - 3$.

Prin urmare $3(n-1) \leq T(n) \leq n^2 + 2n - 3$ adică algoritmul sortării prin inserţie se încadrează în clasele $\Omega(n)$ şi $\mathcal{O}(n^2)$.

4.2.4 Proprietăţi ale sortării prin inserţie

O proprietate importantă este faptul că în cazul şirurilor aproape sortate (număr mic de inversiuni) comportarea algoritmului este apropiată de comportarea din cazul cel mai favorabil. Aceasta înseamnă ca este satisfăcută proprietatea de naturaleţe.

Atât timp cât condiţia de continuare a ciclului interior este $aux < x[j]$ algoritmul de sortare prin inserţie este stabil. Dacă însă se foloseşte $aux \leq x[j]$ atunci stabilitatea nu mai este asigurată.

O altă caracteristică a sortării prin inserţie este faptul că pentru fiecare comparaţie efectuată se realizează deplasarea unui element cu o singură poziţie. Aceasta înseamnă eliminarea a cel mult unei inversiuni, adică aranjarea în ordinea dorită (dar nu neapărat pe poziţiile finale) ale unei perechi de elemente. Cum în cazul cel mai defavorabil, cel al unui şir ordonat descrescător, sunt $n(n-1)/2$ inversiuni rezultă că sunt necesare tot atâtea comparaţii. Aceasta înseamnă ca orice algoritm de sortare bazat pe transformări locale (interschimbări între elemente vecine) aparţine lui $\mathcal{O}(n^2)$. Pentru reducerea ordinului de complexitate ar trebui să se realizeze interschimbări între elemente care nu sunt neapărat vecine. Aceasta este ideea variantei de algoritm descrise în continuare.

4.2.5 Sortare prin inserţie cu pas variabil

Unul dintre dezavantajele sortării prin inserţie este faptul că la fiecare etapă un element al şirului se deplasează cu o singură poziţie. O variantă de reducere a numărului de operaţii efectuate este de a compara elemente aflate la o distanţă mai mare ($h \geq 1$) şi de a realiza deplasarea acestor elemente peste mai multe poziţii. De fapt tehnica se aplică în mod repetat pentru valori din ce în ce mai mici ale pasului h, asigurând *h-sortarea* şirului.

Un şir $x[1..n]$ este considerat h-sortat dacă orice subşir $x[i_0], x[i_0+h], x[i_0+2h]$... este sortat ($i_0 \in \{1, \ldots, h\}$). Aceasta este ideea algoritmului propus de Donald Shell în 1959, algoritm cunoscut sub numele "Shell sort".

Elementul cheie al algoritmului îl reprezintă alegerea valorilor pasului h. Pentru alegeri adecvate ale secvenţei h_k se poate obţine un algoritm de complexitate mai mică (de exemplu $\mathcal{O}(n^{3/2})$ în loc de $\mathcal{O}(n^2)$ cum este în cazul algoritmului clasic de sortare prin inserţie). O astfel de secventă de paşi este $h_k = 2^k - 1$ pentru $1 \leq k \leq \lfloor \lg n \rfloor$.

Exemplu. Exemplificăm ideea algoritmului în cazul unui şir cu 15 elemente pentru următoarele valori ale pasului: $h = 13$, $h = 4$, $h = 1$ (care corespund unui şir h_k dat prin relaţia $h_k = 3h_{k-1} + 1$, $h_1 = 1$). Pentru acest şir s-a ilustrat experimental că este eficient, fără însă a fi determinat teoretic ordinul de complexitate. Procesul de sortare constă în următoarele etape:

Etapa 1: pentru $h = 13$ se aplică algoritmul sortării prin inserţie subşirurilor $(x[1], x[14])$ şi $(x[2], x[15])$, singurele subşiruri cu elemente aflate la distanţa h care au mai mult de un element:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\boxed{14}$	8	10	7	9	12	5	15	2	13	6	1	4	$\boxed{3}$	11

şi se obţine

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

Etapa 2: pentru $h = 4$ se aplică algoritmul sortării prin inserţie succesiv subşirurilor: $(x[1], x[5], x[9], x[13])$, $(x[2], x[6], x[10], x[14])$, $(x[3], x[7], x[11], x[15])$, $(x[4], x[8], x[12])$. După prima subetapă (prelucrarea primului subşir) prin care se ordonează subşirul constituit din elementele marcate:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\boxed{3}$	8	10	7	$\boxed{9}$	12	5	15	$\boxed{2}$	13	6	1	$\boxed{4}$	14	11

se obţine:

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

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

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	$\boxed{8}$	10	7	3	$\boxed{12}$	5	15	4	$\boxed{13}$	6	1	9	$\boxed{14}$	11

obţinându-se aceeaşi configuraţie (subşirul este deja ordonat crescător). Se prelucrează acum cel de al treilea subşir:

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	8	$\boxed{10}$	7	3	12	$\boxed{5}$	15	4	13	$\boxed{6}$	1	9	14	$\boxed{11}$

obţinându-se

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	8	$\boxed{5}$	7	3	12	$\boxed{6}$	15	4	13	$\boxed{10}$	1	9	14	$\boxed{11}$

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

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
2	8	5	$\boxed{7}$	3	12	6	$\boxed{15}$	4	13	10	$\boxed{1}$	9	14	11

obţinându-se

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

Etapa 3: Se aplică sortarea prin inserţie asupra întregului şir. Întrucât şirul la care s-a ajuns prin prelucrările anterioare este aproape sortat numărul de operaţii efectuate în ultima etapă (când se aplică sortarea clasică prin inserţie) este semnificativ mai mic decât dacă s-ar aplica algoritmul de sortare prin inserţie şirului iniţial. Pentru acest exemplu aplicarea directă a sortării prin inserţie conduce la execuţia a 68 de comparaţii pe când aplicarea sortarii cu pas variabil de inserţie conduce la execuţia a 34 de comparaţii. Atât prelucrarea specifică unei etape (pentru o valoare a pasului de inserţie) cât şi întreaga sortare sunt descrise în Algoritmul 4.3.

Algoritmul 4.3 Sortarea prin inserţie cu pas variabil ("Shell sort")

```
inser_pas(real x[1..n], integer h)
integer i, j
real aux
for i ← h + 1, n do
   aux ← x[i]
   j ← i − h
   while j ≥ 1 and aux < x[j] do
      x[j + h] ← x[j]
      j ← j − h
   end while
   x[j + h] ← aux
end for
return x[1..n]
```

```
shellsort(real x[1..n])
integer h
h ← 1
while h ≤ n do
   h ← 3 ∗ h + 1
end while
repeat
   h ← hDIV3
   x[1..n] ← inser_pas(x[1..n], h)
until h = 1
return x[1..n]
```

4.3 Sortare prin selecţie

4.3.1 Principiu

Ideea de bază a acestei metode este:

Pentru fiecare poziţie i, începând cu prima, se selectează din subtabloul ce începe cu acea poziţie cel mai mic element şi se amplasează pe locul respectiv (prin interschimbare cu elementul curent de pe poziţia i)

Structura generală este descrisă în algoritmul `selecţie`.

```
selecţie(real x[1..n])
for i ← 1, n − 1 do
  ⟨ se determină valoarea minimă din x[i..n] şi se interschimbă cu x[i] ⟩
end for
```

Ciclul **for** continuă până la $n-1$ deoarece subtabloul $x[n..n]$ conţine un singur element care este plasat chiar pe poziţia potrivită, ca urmare a interschimbărilor efectuate anterior. O variantă de descriere a metodei selecţiei este Algoritmul 4.4.

Algoritmul 4.4 Sortare prin selecţie

```
selecţie(real x[1..n])
for i ← 1, n − 1 do
  k ← i
  for j ← i + 1, n do
    if x[k] > x[j] then
      k ← j
    end if
  end for
  if k ≠ i then
    x[k] ↔ x[i]
  end if
end for
return (x[1..n])
```

4.3.2 Verificarea corectitudinii

Arătăm că un invariant al ciclului exterior (după i) este: $\{x[1..i-1]$ este ordonat crescător şi $x[i-1] \leq x[j]$ pentru $j = \overline{i,n}\}$. La început $i = 1$ deci $x[1..0]$ este un tablou vid. Ciclul **for** interior (după j) determină poziţia minimului din $x[i..n]$. Aceasta este plasată prin interschimbare pe poziţia i. Se obţine astfel că $x[1..i]$ este ordonat crescător şi că $x[i] \leq x[j]$ pentru $j = \overline{i+1,n}$. După incrementarea lui i (la sfârşitul ciclului după i) se reobţine proprietatea invariantă. La final $i = n$, iar invariantul conduce la $x[1..n-1]$ crescător şi $x[n-1] \leq x[n]$ adică $x[1..n]$ este sortat crescător.

4.3.3 Analiza complexităţii

Indiferent de aranjarea iniţială a elementelor, numărul de comparaţii efectuate este:

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

În cazul cel mai favorabil (şir ordonat crescător) numărul interschimbărilor este $T_M(n) = 0$. În cazul cel mai defavorabil (şir ordonat descrescător) pentru fiecare i se efectuează o interschimbare (trei atribuiri), deci costul corespunzător modificării elementelor tabloului este $T_M(n) = 3\sum_{i=1}^{n-1} 1 = 3(n-1)$. Luând în considerare atât comparaţiile cât şi atribuirile se obţine că algoritmul sortării prin selecţie aparţine clasei $\Theta(n^2)$.

4.3.4 Proprietăţi ale sortării prin selecţie

Algoritmul este parţial natural (numărul de comparaţii nu depinde de gradul de sortare al şirului). În varianta prezentată (când minimul este interschimbat cu poziţia curentă) algoritmul nu este stabil. Dacă însă în locul unei singure interschimbări s-ar realiza deplasarea elementelor subtabloului $x[i..k-1]$ la dreapta cu o poziţie (ca în algoritmul inserţiei), iar $x[k]$ (salvat în prealabil într-o variabilă auxiliară) s-ar transfera in $x[i]$ algoritmul ar deveni stabil.

4.4 Sortare prin interschimbarea elementelor vecine

4.4.1 Principiu

Ideea de bază a acestei metode de sortare este:

> *Se parcurge tabloul de sortat şi se compară elementele vecine iar dacă acestea nu se află în ordinea corectă se interschimbă. Parcurgerea se reia până când nu mai este necesară nici o interschimbare.*

Structura generală este descrisă în algoritmul `interschimbări`.

`interschimbări` (**real** $x[1..n]$)
repeat
  ⟨ se parcurge tabloul şi dacă două elemente vecine nu sunt în ordinea corectă sunt interschimbate ⟩
until ⟨ nu mai este necesară nici o interschimbare ⟩

Să considerăm secvenţa interschimbării elementelor vecine în cazul în care nu sunt în ordinea corectă:

```
for i ← 1, n − 1 do
  if x[i] > x[i + 1] then
    x[i] ↔ x[i + 1]
  end if
end for
```

Folosind ca invariant al prelucrării repetitive proprietatea $\{x[i] \geq x[j], j = \overline{1,i}\}$ se poate arăta că prelucrarea de mai sus conduce la satisfacerea postcondiţiei: $\{x[n] \geq x[i], i = \overline{1,n}\}$. Pentru $i = 1$ proprietatea invariantă este adevărată întrucât $x[1] \geq x[1]$. Presupunem că proprietatea este adevărată pentru i. Dacă $x[i] \leq x[i+1]$ atunci nu se efectuează nici o prelucrare şi rămâne adevărată şi pentru $i + 1$. Dacă în schimb $x[i] > x[i+1]$ atunci se efectuează interschimbarea astfel că $x[i] < x[i+1]$, deci proprietatea devine adevărată şi pentru $i + 1$.

Pe baza acestei proprietăţi a secvenţei de interschimbări se deduce că este suficient să aplicăm această prelucrare succesiv pentru $x[1..n]$, $x[1..n-1]$, ..., $x[1..2]$. Rezultă că o primă variantă a sortării prin interschimbarea elementelor vecine este cea descrisă în Algoritmul 4.5.

Algoritmul 4.5 Sortare prin interschimbarea elementelor vecine

```
interschimbări1(real x[1..n])
for i ← n, 2, −1 do
  for j ← 1, i − 1 do
    if x[j] > x[j + 1] then
      x[j] ↔ x[j + 1]
    end if
  end for
end for
return (x[1..n])
```

4.4.2 Verificarea corectitudinii

Întrucât s-a demonstrat că efectul ciclului interior este că plasează valoarea maximă pe poziţia i rezultă că pentru ciclul exterior poate fi considerată ca invariantă proprietatea $\{x[i+1..n]$ este crescător iar $x[i+1] \geq x[j]$ pentru $j = \overline{1,i}\}$. La ieşirea din ciclul exterior, valoarea contorului i este 1 astfel că se obţine că $x[1..n]$ este ordonat crescător.

4.4.3 Analiza complexităţii

Numărul de comparaţii efectuate nu depinde de gradul de sortare al şirului iniţial fiind în orice situaţie:

$$T_C(n) = \sum_{i=2}^{n}\sum_{j=1}^{i-1} 1 = \sum_{i=2}^{n}(i-1) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}.$$

În schimb, numărul de interschimbări depinde de proprietăţile şirului astfel: în cazul cel mai favorabil (şir sortat crescător) se obţine $T_M(n) = 0$, iar în cazul cel mai defavorabil (şir sortat descrescător) se obţine costul $T_M(n) = 3n(n-1)/2$ (o interschimbare presupune efectuarea a 3 atribuiri). Astfel numărul de prelucrări analizate satisface: $n(n-1)/2 \leq T(n) \leq 2n(n-1)$ adică algoritmul prezentat mai sus aparţine clasei $\Theta(n^2)$.

4.4.4 Variante ale sortării prin interschimbarea elementelor vecine

Algoritmul `interschimbări1` poate fi îmbunătăţit prin reducerea numărului de comparaţii efectuate, în sensul că nu este necesar întotdeauna să se parcurgă tabloul pentru $i = \overline{2, n}$. De exemplu, dacă tabloul este de la început ordonat ar fi suficientă o singură parcurgere care să verifice că nu este necesară efectuarea nici unei interschimbări. Pornind de la această idee se ajunge la varianta descrisă în Algoritmul 4.6.

Algoritmul 4.6 Variantă a algoritmului de sortare prin interschimbarea elementelor vecine

```
interschimbări2(real x[1..n])
repeat
  inter ← false
  for i ← 1, n − 1 do
    if x[i] > x[i + 1] then
      x[i] ↔ x[i + 1]
      inter ← true
    end if
  end for
until inter =false
return x[1..n]
```

Pentru acest algoritm numărul de comparaţii efectuate în cazul cel mai favorabil este $T_C(n) = n - 1$ iar în cazul cel mai defavorabil este $T_C(n) = \sum_{j=1}^{n}(n-1) = n(n-1)$. În ceea ce priveşte costul interschimbărilor acesta este acelaşi ca pentru prima variantă a algoritmului şi anume $0 \leq T_M(n) \leq$

$3n(n-1)/2$. Prin urmare costul total al prelucrărilor analizate satisface:

$$n-1 \leq T(n) \leq \frac{5n(n-1)}{2}$$

adică algoritmul este din $\Omega(n)$ şi $\mathcal{O}(n^2)$.

Întrucât la sfârşitul şirului se formează un subşir crescător rezultă că nu mai este necesar să se facă comparaţii în acea porţiune. Această porţiune este limitată inferior de cel mai mare indice pentru care s-a efectuat interschimbare. Pornind de la această idee se ajunge la varianta descrisă în Algoritmul 4.7.

Algoritmul 4.7 Variantă a algoritmului de sortare prin interschimbarea elementelor vecine

```
interschimbări3(real x[1..n])
m ← n
repeat
  t ← 0
  for i ← 1, m − 1 do
    if x[i] > x[i + 1] then
      x[i] ↔ x[i + 1]
      t ← i
    end if
  end for
  m ← t
until t ≤ 1
return x[1..n]
```

Sortarea prin interschimbarea elementelor vecine asigură, la fiecare pas al ciclului exterior, plasarea câte unui element (de exemplu maximul din subtabloul tratat la etapa respectivă) pe poziţia finală. O variantă ceva mai eficientă ar fi ca la fiecare etapă să se plaseze pe poziţiile finale câte două elemente (minimul respectiv maximul din subtabloul tratat la etapa respectivă). Pe de altă parte prin reţinerea indicelui ultimei interschimbări efectuate, atât la parcurgerea de la stânga la dreapta cât şi de la dreapta la stânga se poate limita regiunea analizată cu mai mult de o poziţie atât la stânga cât şi la dreapta (când tabloul conţine porţiuni deja sortate). Această variantă (cunoscută sub numele de "shaker sort") este descrisă în Algoritmul 4.8.

Atâta timp cât condiţia pentru interschimbare este specificată prin inegalitate strictă ($x[i] > x[i+1]$) oricare dintre variantele algoritmului este stabilă.

Algoritmul 4.8 Varianta "shaker sort"

```
shakersort(real x[1..n])
integer s, d, i, t
s ← 1; d ← n
repeat
  t ← 0
  for i ← s, d − 1 do
    if x[i] > x[i + 1] then
      x[i] ↔ x[i + 1]; t ← i
    end if
  end for
  if t ≠ 0 then
    d ← t; t ← 0
    for i ← d, s + 1, −1 do
      if x[i] < x[i − 1] then
        x[i] ↔ x[i − 1]; t ← i
      end if
    end for
    s ← t
  end if
until t = 0 or s = d
return x[1..n]
```

4.5 Limite ale eficienţei algoritmilor de sortare bazaţi pe compararea între elemente

Valorile comparative ale numărului de prelucrări (comparaţii şi mutări asupra elementelor tabloului) pentru algoritmii de sortare descrişi sunt prezentate în Tabelul 4.1. În secţiunea referitoare la sortarea prin inserţie s-a menţionat faptul că algoritmii de sortare bazaţi pe transformări locale (de exemplu, interschimbări ale elementelor vecine) au o complexitate pătratică în cazul cel mai defavorabil.

În cazul general al algoritmilor de sortare ce folosesc compararea între elemente se poate arăta că ordinul de complexitate este cel puţin $n \lg n$. Acest lucru poate fi justificat prin faptul că procesul de sortare poate fi ilustrat printr-un arbore de decizie caracterizat prin faptul că fiecare nod intern corespunde unei comparaţii între două elemente iar frontiera conţine noduri corespunzătoare tuturor permutărilor posibile ale mulţimii elementelor ce trebuie sortate. Figura 4.1 ilustrează arborele de decizie corespunzător sortării prin inserţie în cazul unui tablou cu 3 elemente. Ordinul de complexitate al algoritmului este dat de adâncimea arborelui (numărul de nivele). În cazul unei mulţimi cu n elemente, numărul permutărilor este $n!$ iar întrucât arborele de decizie este binar

Algoritm	Caz favorabil		Caz defavorabil	
	$T_C(n)$	$T_M(n)$	$T_C(n)$	$T_M(n)$
inserţie1	$n-1$	$2(n-1)$	$\frac{n^2+n-2}{2}$	$\frac{n^2+3n-4}{2}$
	$3(n-1) \leq T(n) \leq n^2+2n-3$, $(\Omega(n), O(n^2))$			
selecţie	$\frac{n(n-1)}{2}$	0	$\frac{n(n-1)}{2}$	$3(n-1)$
	$\frac{n(n-1)}{2} \leq T(n) \leq \frac{n^2+5n-6}{2}$, $(\Theta(n^2))$			
interschimbări1	$\frac{n(n-1)}{2}$	0	$\frac{n(n-1)}{2}$	$3\frac{n(n-1)}{2}$
	$\frac{n(n-1)}{2} \leq T(n) \leq 2n(n-1)$, $(\Theta(n^2))$			
interschimbări2	$n-1$	0	$n(n-1)$	$3\frac{n(n-1)}{2}$
	$n-1 \leq T(n) \leq \frac{5n(n-1)}{2}$,$(\Omega(n), O(n^2))$			

Tabelul 4.1: Costurile algoritmilor elementari de sortare (număr de comparaţii si de transferuri de elemente) şi ordinele de complexitate corespunzătoare

rezultă că adâncimea lui, k, trebuie să satisfacă: $1+2+\ldots+2^k \geq n!$ adică $2^{k+1}-1 \geq n!$ ceea ce implică faptul că $k \in \Omega(\lg n!) = \Omega(n \lg n)$. Prin urmare, numărul de comparaţii efectuate, în cel mai defavorabil caz, de către un algoritm de sortare bazat pe compararea elementelor este cel puţin $n \lg n$. Un exemplu de algoritm care atinge această limită inferioară este cel bazat pe utilizarea unei structuri de date speciale numită "heap". Pe lângă acest algoritm (descris în secţiunea următoare) există şi alţi algoritmi care au ordin de complexitate $n \lg n$ fie în cazul cel mai defavorabil (sortarea prin interclasare) sau în cazul mediu (sortarea rapidă). Aceşti algoritmi sunt prezentaţi în capitolul dedicat tehnicii divizării.

4.6 Sortarea folosind structura de tip "heap"

Un "heap", numit uneori *ansamblu* sau *movilă* este un tablou de elemente $H[1..n]$ care are proprietatea că pentru fiecare element $H[i]$ cu $1 \leq i \leq \lfloor n/2 \rfloor$ sunt adevărate inegalităţile: $H[i] \geq H[2i]$ şi $H[i] \geq H[2i+1]$ (în cazul în care $2i+1 \leq n$. O astfel de structură poate fi vizualizată sub forma unui arbore binar aproape complet (arbore în care fiecare nod are doi fii, iar fiecare nivel cu excepţia ultimului este complet). Fii unui nod aflat pe poziţia i în tablou se află pe poziţiile $2i$ (fiul stâng) respectiv $2i+1$ (fiul drept). In baza aceleiaşi proprietăţi părintele nodului de pe poziţia i în tablou se află pe poziţia $\lfloor i/2 \rfloor$. Câteva exemple de astfel de structură vizualizată arborescent sunt prezentate

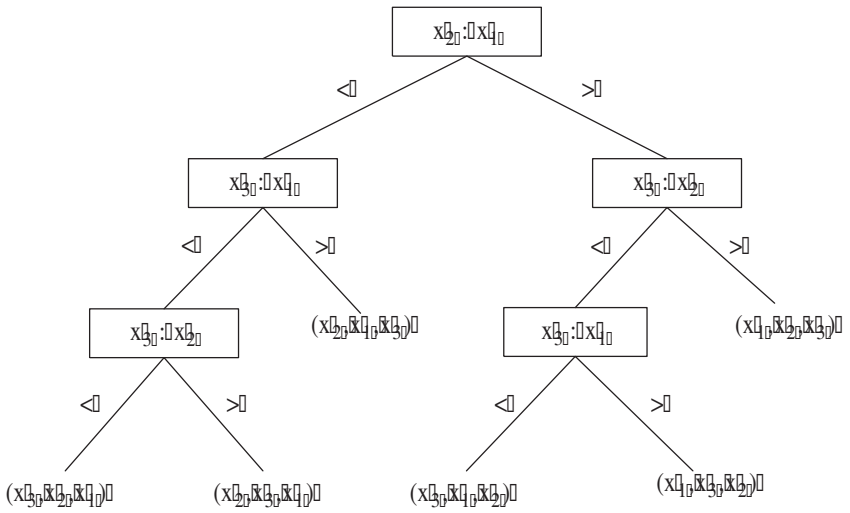

Figura 4.1: Arbore de decizie corespunzător sortării prin inserţie în cazul unui tablou cu trei elemente distincte

în Figura 4.2.

Aceste structuri de date sunt utile atât pentru sortarea eficientă a unui tablou cât şi pentru implementarea cozilor de priorităţi. În această secţiune structura de tip "heap" este folosită doar pentru implementarea unei metode de sortare.

Una dintre cele mai importante prelucrări referitoare la un heap este aceea de a asigura satisfacerea proprietăţii corespunzătoare fiecărui element (nod): valoarea corespunzătoare nodului să fie mai mare decât valorile corespunzătoare fiilor. Metoda de transformare a unui heap astfel încât nodul i să satisfacă proprietatea specifică (în ipoteza că subarborii având rădăcinile în cei doi fii satisfac fiecare proprietatea de heap) este descrisă în Algoritmul 4.9 care primeşte ca date de intrare tabloul corespunzător heapului şi indicele nodului de unde începe procesul de "reparare". Trebuie precizat faptul că nu toate cele n elemente ale tabloului sunt considerate active la un moment dat astfel ca dimensiunea heapului, specificată în algoritm prin $size(H)$ poate fi mai mică decât n.

Intrucât un subarbore are cel mult $2n/3$ noduri (cazul cel mai defavorabil este întâlnit când ultimul nivel al arborelului este completat pe jumătate) numărul de comparaţii efectuate, $T(n)$, satisface $T(n) \leq T(n/3) + 2$ ceea ce conduce la faptul că $T(n) \in \mathcal{O}(\lg n)$.

Procesul propriu-zis de sortare constă în două etape principale:

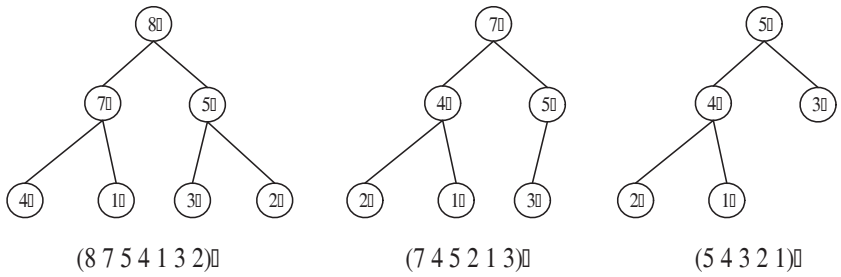

Figura 4.2: Exemple de heap-uri descrise arborescent

Algoritmul 4.9 Restaurarea proprietăţii de heap pornind de la nodul i

```
reHeap(H[1..n],i)
if 2i ≤ size(H) and H[2i] > H[i] then
   imax ← 2i
else
   imax ← i
end if
if 2i + 1 ≤ size(H) and H[2i + 1] > H[imax] then
   imax ← 2i + 1
end if
if imax ≠ i then
   H[i] ↔ H[imax]
   H[1..n] ←reHeap(H,imax)
end if
return  H[1..n]
```

- Construirea, pornind de la tabloul iniţial, al unui heap. Procesul de construire se bazează pe utilizarea algoritmului `reHeap` pentru fiecare element din prima jumătate a tabloului începând cu elementul de pe poziţia $\lfloor n/2 \rfloor$
- Eliminarea succesivă din heap a nodului rădăcină şi plasarea acestuia la sfârşitul tabloului corespunzător heap-ului. La fiecare etapă dimensiunea heapului scade cu un element iar subtabloul ordonat de la sfârşitul zonei corespunzătoare tabloului iniţial creşte cu un element.

Cele două etape sunt descrise în Algoritmul 4.10.

Intrucât algoritmul `heapSort` apelează de $n-1$ ori algoritmul `reHeap` care are ordinul de complexitate $\mathcal{O}(\lg\ n)$ iar cum algoritmul de construire are ordinul $\mathcal{O}(n \lg n)$ rezultă că `heapSort` are de asemenea ordinul $\mathcal{O}(n \lg n)$. Merită menţionat faptul că marginea superioară $n \lg n$ pentru algoritmul de construire

Algoritmul 4.10 Construirea unui heap şi sortarea pe baza acestuia

```
construireHeap(H[1..n])
size(H) ← n
for i ← ⌊n/2⌋, 1, −1 do
   H[1..n] ← reHeap(H,i)
end for
return H[1..n]
```

```
heapSort(H[1..n])
H[1..n] ←construireHeap(H)
for i ← n, 2, −1 do
   H[i] ↔ H[1]
   size(H) ← size(H) − 1
   H[1..n] ←reHeap(H,1)
end for
return H[1..n]
```

al unui heap este largă şi că se poate demonstra că acesta are de fapt complexitate liniară.

4.7 Algoritmi de sortare care nu folosesc compararea între elemente

Dacă se cunosc informaţii suplimentare despre elementele şirului de sortat (de exemplu că aparţin unei mulţimi de forma $\{1, \ldots, m\}$) atunci se poate evita compararea directă între elemente şi în felul acesta se pot obţine algoritmi de complexitate mai mică decât $n\lg n$.

4.7.1 Sortarea folosind tabel de frecvenţe

Să considerăm problema sortării unui tablou $x[1..n]$ având elemente din $\{1, 2, \ldots, m\}$. Pentru acest caz particular poate fi utilizat un algoritm de complexitate $\mathcal{O}(m + n)$ bazat pe următoarele idei:

- se construieşte tabelul $f[1..m]$ al frecvenţelor de apariţie ale valorilor elementelor tabloului $x[1..n]$;
- se calculează frecvenţele cumulate asociate;
- se foloseşte tabelul frecvenţelor cumulate pentru a construi tabloul ordonat y.

Ideea fundamentală a acestui algoritm este de a determina, pentru fiecare element al tabloului de sortat, câte dintre elemente sunt mai mici decât el. Din acest motiv metoda este cunoscută şi sub numele de sortare prin numărare ("counting sort"). Acest lucru este făcut însă calculând frecvenţe cumulate şi fără a compara direct elementele între ele. O variantă de implementare este descrisă în Algoritmul 4.11. Dacă $m \in \Theta(n)$ atunci algoritmul are complexitate liniară. Dacă însă m este mult mai mare decât n (de exemplu $m \in \Theta(n^2)$)

atunci ordinul de complexitate al algoritmului de sortare este determinat de m.

În ceea ce priveşte stabilitatea, atât timp cât ciclul de construire al tabloului y este descrescător, algoritmul este stabil. Spre deosebire de ceilalţi algoritmi de sortare analizaţi până acum, acesta utilizează spaţiu de memorie adiţional de dimensiune cel puţin n.

Algoritmul 4.11 Sortare pe baza unui tablou de frecvenţe

```
CountingSort(integer x[1..n],m)
integer i, f[1..m], y[1..n]
for i ← 1, m do
   f[i] ← 0
end for
for i ← 1, n do
   f[x[i]] ← f[x[i]] + 1
end for
for i ← 2, m do
   f[i] ← f[i − 1] + f[i]
end for
for i ← n, 1, −1 do
   y[f[x[i]]] ← x[i]
   f[x[i]] ← f[x[i]] − 1
end for
for i ← 1, n do
   x[i] ← y[i]
end for
return x[1..n]
```

4.7.2 Sortarea pe baza cifrelor

În cazul în care elementele tabloului de sortat sunt numere naturale cu cel mult k cifre, iar k este mic în raport cu n atunci sortarea poate fi realizată în timp liniar în raport cu n. Ideea acestei metode este de a realiza succesiv sortarea la nivelul cifrelor numărului pornind de la cifra cea mai puţin semnificativă: se ordonează tabloul în raport cu cifra cea mai puţin semnificativă a fiecarui număr (folosind de exemplu sortarea pe baza tabelului de frecvenţe) după care se sortează în raport cu cifra de rang imediat superior ş.a.m.d. până se ajunge la cifra cea mai semnificativă. Algoritmul 4.12 descrie structura generală precum şi adaptarea algoritmului `CountingSort` în cazul sortării pe baza cifrelor (funcţia `cs`).

Algoritmul `cs` se apelează pentru $m = 9$ deci este de ordinul $\mathcal{O}(n)$. Pe de altă parte pentru ca algoritmul de sortare pe baza cifrelor să funcţioneze

Algoritmul 4.12 Sortarea pe baza cifrelor

```
radixsort(integer x[1..n], k)
integer i
for i ← 0, k − 1 do
   x[1..n] ← cs(x[1..n], 9, i)
end for
return x[1..n]
```

```
cs(integer x[1..n],m,c )
integer i, j, f[0..m], y[1..n]
for i ← 0, m do
   f[i] ← 0
end for
for i ← 1, n do
   j ← (x[i]DIV putere(10, c))MOD 10
   f[j] ← f[j] + 1
end for
for i ← 1, m do
   f[i] ← f[i − 1] + f[i]
end for
for i ← n, 1, −1 do
   j ← (x[i]DIV putere(10, c))MOD 10
   y[f[j]] ← x[i]
   f[j] ← f[j] − 1
end for
for i ← 1, n do
   x[i] ← y[i]
end for
return x[1..n]
```

corect este necesar ca subalgoritmul de sortare prin numărare apelat să fie stabil. Dacă $m = 10^k$ este semnificativ mai mare decât n atunci algoritmul nu este eficient. Dacă însă k este mult mai mic decât n atunci un algoritm de complexitate $\mathcal{O}(kn)$ ar putea fi acceptabil.

4.8 Probleme

Problema 4.1 Se consideră un tablou ale cărui elemente conţin două tipuri de informaţii: nume şi nota. Să se ordoneze descrescător după notă, iar pentru aceeaşi notă crescător după nume.

Indicaţie. Se sortează tabloul crescător după nume, după care se aplică un algoritm stabil pentru sortarea descrescătoare după notă.

Problema 4.2 Propuneţi un algoritm care sortează crescător elementele de pe poziţiile impare ale unui tablou şi descrescător cele de pe poziţii pare.

Indicaţie. Se aplică ideea de la h-sortarea prin inserţie pentru $h = 2$ însă în prima etapă se sortează crescător iar în a doua se sortează descrescător.

Problema 4.3 Se consideră o matrice pătratică de dimensiune n şi elemente

reale, a_{ij}. Să se reorganizeze matricea prin interschimbări de linii şi coloane astfel încât elementele diagonalei principale să fie ordonate crescător.

Indicaţie. Se aplică algoritmul de sortare prin selecţie asupra şirului (a_{11}, a_{22}, ..., a_{nn}) însă interschimbarea elementului a_{ii} cu elementul a_{jj} se realizează prin interschimbarea intregii linii i cu linia j şi a coloanei i cu coloana j.

Problema 4.4 Se consideră algoritmul:

```
alg(x[1..n])
for i ← 1, n − 1 do
  if i MOD 2 = 1 then
    for j ← 1, n − 1, 2 do
      if x[j] > x[j + 1] then
        x[j] ↔ x[j + 1]
      end if
    end for
  else
    for j ← 2, n − 1, 2 do
      if x[j] > x[j + 1] then
        x[j] ↔ x[j + 1]
      end if
    end for
  end if
end for
return x[1..n]
```

Să se stabilească care este efectul algoritmului asupra tabloului $x[1..n]$ şi să se stabilească ordinul de complexitate.

Indicaţie. Algoritmul asigură ordonarea crescătoare a lui $x[1..n]$ iar ordinul de complexitate este $\mathcal{O}(n^2)$.

Problema 4.5 Să se adapteze algoritmul de sortare pe baza tabelului de frecvenţe (Algoritmul 4.11) astfel încât tabloul y să conţine elementele lui x în ordine descrescătoare.

Indicaţie. Atribuirea $y[f[x[i]]] \leftarrow x[i]$ se înlocuieşte cu $y[n+1-f[x[i]]] \leftarrow x[i]$, iar pentru a asigura stabilitatea parcurgerea lui $x[1..n]$ se va face începând cu primul element.

Problema 4.6 Se consideră un set de valori din $\{1, \ldots, m\}$. Preprocesaţi acest set folosind un algoritm de complexitate $\mathcal{O}(\max\{m, n\})$ astfel încât răspunsul la întrebarea "câte elemente se află în intervalul $[a, b]$?", $a, b \in \{1, \ldots, m\}$ să poată fi obţinut în timp constant.

Indicaţie. În tabelul frecvenţelor cumulate (care poate fi construit in $\mathcal{O}(m+n)$) elementul $f[b]$ reprezintă numărul de elemente din tablou mai mici sau egale decât b, iar $f[a-1]$ reprezintă numărul elementelor strict mai mici decât a.

Astfel numărul de elemente cuprinse în $[a, b]$ se poate obţine printr-o singură operaţie: $f[b] - f[a-1]$.

Problema 4.7 Propuneţi un algoritm de complexitate liniară pentru ordonarea crescătoare a unui tablou constituit din n valori întregi aparţinând mulţimii $\{0, 1, \ldots, n^2 - 1\}$.

Indicaţie. Se vor interpreta valorile ca fiind reprezentate în baza n. Astfel fiecăreia dintre cele n valori îi corespunde o pereche de "cifre", c_0,$c_1 \in \{0, \ldots, n-1\}$ obţinute ca rest respectiv cât al împărţirii la n. Se poate astfel aplica algoritmul sortării pe baza cifrelor în cazul în care valoarea maximă a "cifrelor" este $m = n - 1$ iar numărul de "cifre" este $k = 2$. Ordinul de complexitate este în acest caz $\mathcal{O}(k(m+n))$ adică $\mathcal{O}(n)$.

Problema 4.8 Se consideră un tablou constituit din triplete de trei valori întregi corespunzătoare unei date calendaristice (zi,luna,an). Propuneţi un algoritm eficient de ordonare crescătoare după valoarea datei.

Indicaţie. Se aplică un algoritm stabil de complexitate liniară (de exemplu sortarea prin numărare) succesiv pentru zi, luna şi an.