Tehnici de sortare

Cuprins

Tehnici de sortare

Sortarea este o operaţie fundamentala în informatică, mulţi algoritmi o folosesc ca pas intermediar, are o largă aplicabilitate în informatică şi disciplinele conexe. În liceu, la disciplina informatică se studiază câteva metode de sortare, dar acestea sunt reprezentative şi apar în toţi anii de studiu, ceea ce le conferă o mare importanţă.

Intrare: Un şir de *n* numere $(a_1, a_2, \ldots, a_n)$.

Ieşire: O permutare (reordonare) $(a'_1, a'_2, \ldots, a'_n)$ a şirului dat astfel încât $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.

Şirul de intrare este, de obicei, un tablou cu *n* elemente, deşi el poate fi reprezentat şi în alt mod, de exemplu sub forma unei liste înlănţuite.

În practică, numerele care trebuie ordonate sunt rareori valori izolate. De obicei, fiecare număr face parte dintr-o colecţie de date numita *articol*. Fiecare articol conţine o *cheie*, care este valoarea ce trebuie ordonată, iar restul articolului conţine *date adiţionale*, care sunt, de obicei, mutate împreuna cu cheia. În practică, atunci când un algoritm de ordonare interschimbă cheile, el trebuie să interschimbe şi datele adiţionale. Dacă fiecare articol include o cantitate mare de date adiţionale, de multe ori interschimbăm un tablou de pointeri la articole, în loc să interschimbăm articolele înseşi, cu scopul de a minimiza cantitatea de date care este manevrată.

Dintr-un anumit punct de vedere, tocmai aceste detalii de implementare disting un algoritm de programul propriu-zis. Faptul că sortam numere individuale sau articole mari, ce conţin numere, este irelevant pentru *metoda* prin care o procedura de ordonare determină ordinea elementelor. Astfel, atunci când ne concentrăm asupra problemei sortării, presupunem, de obicei, că intrarea constă numai din numere. Implementarea unui algoritm care sortează numere este directă din punct de vedere conceptual, deşi, într-o situaţie practică dată, pot exista şi alte subtilităţi care fac ca implementarea propriu-zisă a algoritmului să fie o sarcină dificilă.

Au fost inventaţi un număr foarte mare de algoritmi diferiţi pentru sortarea unui vector, însă, deşi diferiţi, sunt legaţi între ei şi nu sunt greu de învăţat. În cartea lui *D. Knuth* „Arta programării calculatoarelor”, vol. III (*Căutări şi ordonări*) sunt

prezentate 33 de metode de ordonare. O parte vor fi prezentate în această lucrare. Este indicat să cunoaştem caracteristicile fiecărei metode de sortare, ca să fim în măsură să facem alegeri în cunoştinţă de cauză pentru aplicaţii particulare.

Tehnicile de sortare se pot clasifica în mai multe feluri, în funcţie de complexitate, de cantitatea de memorie necesară. Iată câteva clasificări:

După cantitatea de memorie utilizată:

a) metode directe (sortează vectorul în spaţiul său de memorie)

b) metode indirecte (necesită vector auxiliar)

După spaţiul de memorie utilizat:

a) metode interne (ordonează tablouri în memoria internă)

b) metode externe (ordonează sau interclasează fişiere din memoria externă)

După ordinul de complexitate:

a) metode liniare

b) metode pătratice

c) metode n $\log_2$ n

O altă clasificare poate fi făcută după dificultatea algoritmilor implicaţi:

a) metode simple - se bazează pe algoritmi de dificultate redusă (inserţie, selecţie, bubblesort), dar mai puţin eficiente.

b) metodele avansate - se bazează pe algoritmi puţin mai complicaţi, care necesită mici "artificii" pentru a realiza ordonarea (quicksort, sortarea prin interclasare, heap-sort), dar care sunt mai eficiente decât cele directe.

1. Sortarea prin inserţie

Începem cu *sortarea prin inserţie*, care este un algoritm eficient pentru sortarea unui număr mic de obiecte. Sortarea prin inserţie funcţionează în acelaşi fel în care mulţi oameni sortează un pachet de cărţi de joc. Se începe cu pachetul aşezat pe masa cu faţa în jos şi cu mâna stânga goala. Apoi, luam câte o carte de pe masa şi o inseram în poziţia corectă în mâna stânga. Pentru a găsi poziţia corecta pentru o carte data, o comparăm cu fiecare dintre cărţile aflate deja în mâna stânga, de la dreapta la stânga (sau de la stânga la dreapta), aşa cum este ilustrat în Figura 1.1.

Figura 1.1 Modul de ordonare a cărţilor, folosind metoda sortării prin inserţie.

Pseudocodul pentru sortarea prin inserţie este prezentat ca o procedura numita *Sorteaza-Prin-Inserţie*, care are ca parametru un vector $A[1..n]$ conţinând un sir de lungime n care urmează a fi sortat. (Pe parcursul codului, numărul de elemente ale lui A este notat prin *lungime*$[A]$.) Numerele de intrare sunt *sortate pe loc*, în cadrul aceluiaşi vector A, cel mult un număr constant dintre acestea sunt memorate în zone de memorie suplimentare. Când *Sorteaza-Prin-Inserţie* se termina, vectorul iniţial A va conţine elementele şirului de ieşire sortat.

```
Subalgoritm Sorteaza-Prin-Inserţie(A)
1: pentru j ← 2, lungime[A] executa
2:     cheie ← A[j]
3:     i ← j - 1
4:     cât timp i > 0 şi A[i] > cheie executa
5:         A[i + 1] ← A[i]
6:         i ← i – 1
7:     A[i + 1] ← cheie  {Insereaza A[j] în sirul sortat A[1..j - 1]}
```

Figura 1.2 ilustrează modul de funcţionare a acestui algoritm pentru $A = (5, 2, 4, 6, 1, 3)$. Indicele j corespunde "cărţii" care urmează a fi inserată în mâna stânga. Elementele $A[1..j - 1]$ corespund mulţimii de cărţi din mâna, deja sortate, iar elementele $A[j + 1..n]$ corespund pachetului de cărţi aflate încă pe masă. Indicele se deplasează de la stânga la dreapta în interiorul vectorului. La fiecare iteraţie, elementul $A[j]$ este ales din vector (linia 2). Apoi, plecând de la poziţia $j - 1$, elementele sunt, succesiv, deplasate o poziţie spre dreapta până când este găsita poziţia corecta pentru $A[j]$ (liniile 3–6), moment în care acesta este inserat (linia 7).

a_1	a_2	a_3	a_4	a_5	a_5
5	2	4	6	1	3
2	5	4	6	1	3
2	4	5	6	1	3
2	4	5	6	1	3
1	2	4	5	6	3
1	2	3	4	5	6

Figura 1.2 Modul de operare a procedurii Sorteaza-Prin-Inserţie asupra vectorului *A =(*5, 2, 4, 6, 1, 3*)*.

Poziţia indicelui *j* este indicata prin colorarea celulei corespunzătoare din tabelul din figură.

La scrierea pseudocodului vom folosi următoarele convenţii:

1. Indentarea indica o structura de bloc (pentru instrucţiunile ***daca***, ***pentru***, ***repeta*** şi ***cat timp***). De exemplu, corpul ciclului pentru, care începe în linia 1, consta din liniile 2–7 iar corpul ciclului cât timp, care începe în linia 4, conţine liniile 5 şi 6, dar nu şi linia 7. Acest mod de indentare se aplica şi structurilor de tipul ***daca-atunci-altfel***.

2. Ciclurile de tipul ***cât timp***, ***pentru***, ***repeta*** şi construcţiile condiţionale ***daca***, ***atunci*** şi ***altfel*** au aceeaşi interpretare ca şi structurile similare din Pascal.

3. Comentariile se precizează între paranteze {}, la fel ca în C/C++.

4. O atribuire multipla de forma $i \leftarrow j \leftarrow e$ înseamnă atribuirea valorii expresiei *e* ambelor variabile *i* şi *j*, aceasta ar trebui tratată ca un echivalent al atribuirii $j \leftarrow e$, urmată de atribuirea $i \leftarrow j$.

5. Variabilele (de exemplu *i*, *j*, *cheie*) sunt locale pentru o procedura dată sau un subalgoritm. Nu vom utiliza variabile globale fără a preciza acest lucru în mod explicit.

6. Elementele unui vector sunt accesate specificând numele vectorului urmat de indice în paranteze drepte. De exemplu, *A*[*i*] indica elementul de rang *i* al vectorului *A*. Notaţia ".." este folosita pentru a indica un domeniu de valori în cadrul unui vector. Astfel, *A*[1..*j*] indica un subvector al lui *A* constând din elementele *A*[1],*A*[2], ...,*A*[*j*].

7. Datele compuse sunt, în mod uzual, organizate în *obiecte* care conţin *atribute* sau *câmpuri*.

Un anumit câmp este accesat folosind numele câmpului urmat de numele obiectului său în paranteze drepte. De exemplu, tratam un vector ca pe un obiect cu atributul *lungime* indicând numărul de elemente ale acestuia. Pentru a specifica numărul de elemente ale unui vector *A*, se va scrie *lungime* [*A*]. Deşi vom folosi parantezele drepte atât pentru indexarea elementelor unui vector, cât şi pentru atributele obiectelor, va fi clar din context care este interpretarea corecta.

O variabilă reprezentând un vector sau un obiect este tratata ca un pointer spre datele care reprezintă vectorul sau obiectul. Pentru toate câmpurile *f* ale unui obiect *x*, atribuirea $y \leftarrow x$ are ca efect $f[y] = f[x]$. Mai mult, daca acum avem $f[x] \leftarrow 3$, atunci nu numai $f[x] = 3$, dar, în acelaşi timp, avem şi $f[y] = 3$. Cu alte cuvinte, *x* şi *y* indica spre (sau "sunt") acelaşi obiect dupa atribuirea $y \leftarrow x$. Uneori, un pointer nu se va referi la nici un obiect. În acest caz special pointerul va primi valoarea nil (null sau 0).

8. Parametrii sunt transmişi unei proceduri *prin valoare*: procedura apelata primeşte propria sa copie a parametrilor şi, daca atribuie o valoare unui parametru, schimbarea *nu* este văzuta de procedura apelanta. Când obiectele sunt transmise procedurii, este copiat doar pointerul spre datele reprezentând obiectul, nu şi câmpurile acestuia. De exemplu, daca *x* este un parametru al unei proceduri apelate, atribuirea $x \leftarrow y$ în cadrul procedurii apelate nu este vizibila din procedura apelanta. Atribuirea $f[x] \leftarrow 3$ este, totuşi, vizibila.

Analiza sortării prin inserţie

Timpul de execuţie necesar procedurii Sorteaza-Prin-Inserţie depinde de intrare: sortarea a o mie de numere ia mai mult timp decât sortarea a trei. Mai mult decât atât, Sorteaza-Prin-Inserţie poate să consume timpi diferiţi pentru a sorta două şiruri de numere de aceeaşi dimensiune, în funcţie de măsura în care acestea conţin numere aproape sortate. În general, timpul necesar unui algoritm creşte odată cu dimensiunea datelor de intrare, astfel încât este tradiţional să se descrie timpul de execuţie al unui program în funcţie de dimensiunea datelor de intrare. În acest scop,

trebuie să definim cu mai multă precizie termenii de “timp de execuţie” şi “dimensiune a datelor de intrare”.

Definiţia *dimensiunii datelor de intrare* depinde de problema studiată. Pentru multe probleme, inclusiv sortarea, cea mai naturală măsură este *numărul de obiecte din datele de intrare* – de exemplu, pentru sortare, un vector de dimensiune n. Pentru multe alte probleme, ca spre exemplu înmulţirea a doi întregi, cea mai buna măsură pentru dimensiunea datelor de intrare este *numărul total de biţi* necesari pentru reprezentarea datelor de intrare în notaţie binară. Uneori, este mai potrivit să exprimăm dimensiunea datelor de intrare prin două numere în loc de unul. De exemplu, dacă datele de intrare ale unui algoritm sunt reprezentate de un graf, dimensiunea datelor de intrare poate fi descrisa prin numărul de vârfuri şi muchii ale grafului. În cazul sortării unui vector de numere, putem descrie dimensiunea datelor prin numărul de valori şi prin mărimea acestora ca numere. Pentru fiecare problema pe care o vom studia, vom indica măsura utilizată pentru dimensiunea datelor de intrare.

Timpul de execuţie a unui algoritm pentru un anumit set de date de intrare este determinat de numărul de operaţii primitive sau “paşi” executaţi. Este util să definim noţiunea de “pas” astfel încât să fie cât mai independent de calculator. Pentru execuţia unei linii din pseudocod este necesară o durată constantă de timp. O anumita linie poate avea nevoie de un timp de execuţie diferit decât o alta, dar vom presupune că fiecare execuţie a liniei i consuma timpul c_i, unde c_i este o constantă. Acest punct de vedere este conform cu modelul RAM şi, în acelaşi timp, reflectă, destul de bine, modul în care pseudocodul poate fi, de fapt, utilizat în cele mai multe cazuri concrete.

În prezentarea care urmează, expresia noastră pentru timpul de execuţie al algoritmului Sorteaza-Prin-Inserţie va evolua de la o formulă relativ complicată, care foloseşte toate costurile de timp c_i, la una mult mai simplă în notaţii, care este mai concisă şi mai uşor de manevrat. Aceasta notaţie mai simplă va face, de asemenea, uşor de determinat dacă un algoritm este mai eficient decât altul.

Începem prin a relua prezentarea procedurii Sorteaza-Prin-Inserţie, adăugând “costul” de timp pentru fiecare instrucţiune şi un număr care reprezintă de

câte ori aceasta este efectiv executata. Pentru fiecare $j = 2, 3, \ldots, n$, unde $n = [A]$, vom nota cu t_j numărul de execuţii ale testului cât timp din linia 5 pentru valoarea fixata j. Vom presupune că un comentariu nu este o instrucţiune executabila, prin urmare nu cere timp de calcul.

Subalgoritm Sorteaza-Prin-Inserţie(A)	*cost*	*timp*
1: pentru $j \leftarrow 2$, *lungime*[A] executa	c_1	n
2: *cheie* $\leftarrow A[j]$	c_2	n-1
3: $i \leftarrow j - 1$	c_3	n-1
4: cât timp $i > 0$ şi $A[i] >$ *cheie* executa	c_4	$\Sigma^n_{j=2} t_j$
5: $A[i + 1] \leftarrow A[i]$	c_5	$\Sigma^n_{j=2}(t_j-1)$
6: $i \leftarrow i - 1$	c_6	$\Sigma^n_{j=2}(t_j-1)$
7: $A[i + 1] \leftarrow$ *cheie*	c_7	n-1

Timpul de execuţie al algoritmului este suma tuturor timpilor de execuţie corespunzători fiecărei instrucţiuni executate: o instrucţiune care consuma timpul c_i pentru execuţie şi este executată de n ori, va contribui cu $c_i n$ la timpul total de execuţie. Pentru a calcula $T(n)$,timpul de execuţie pentru Sorteaza-Prin-Inserţie, vom aduna produsele mărimilor indicate în coloanele *cost* şi *timp*, obţinând

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

Chiar pentru date de intrare de aceeaşi mărime, timpul de execuţie al unui algoritm dat poate să depindă de *conţinutul* datelor de intrare. De exemplu, pentru Sorteaza-Prin-Inserţie, cazul cel mai favorabil apare când vectorul de intrare este deja sortat. Pentru fiecare $j = 2, 3, \ldots n$,vom găsi că $A[i] \leq$ *cheie* în linia 4, când i are valoarea iniţiala $j - 1$. Rezulta $t_j = 1$ pentru $j = 2, 3, \ldots, n$ şi timpul de execuţie în cazul cel mai favorabil este

$T(n) = c1n + c2(n - 1) + c3(n - 1) + c4(n - 1) + c7(n - 1)$

$= (c1 + c2 + c3 + c4 + c7)n - (c2 + c3 + c4 + c7)$

Acest timp de execuţie poate fi exprimat sub forma $an + b$ pentru anumite *constante* a şi b care depind doar de timpii de execuţie c_i, fiind astfel o *funcţie liniara* de n.

Daca vectorul este sortat în ordine inversa – adică, în ordine descrescătoare – obţinem cazul cel mai defavorabil. În aceasta situaţie trebuie să comparăm fiecare

element $A[j]$ cu fiecare element din subvectorul $A[1..j - 1]$, şi, astfel, $t_j = j$ pentru $j = 2, 3, \ldots, n$. Observând că

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

şi

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

Găsim că în cazul cel mai defavorabil timpul de execuţie pentru Sorteaza-Prin-Inserţie este

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) + c_6\left(\frac{n(n-1)}{2}\right) \\ &+ c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\ &- (c_2 + c_4 + c_5 + c_8)\,. \end{aligned}$$

Rezulta că timpul de execuţie în cazul cel mai defavorabil poate fi exprimat sub forma $an^2 + bn + c$, unde constantele a, b şi c depind, din nou, de costurile c_i ale instrucţiunilor, fiind astfel o *funcţie pătratica* de n.

În analiza sortării prin inserţie am cercetat ambele situaţii extreme: cazul cel mai favorabil, în care vectorul de intrare era deja sortat, respectiv cel mai defavorabil, în care vectorul de intrare era sortat în ordine inversa. În continuare (pe tot parcursul acestei lucrări), ne vom concentra, de regula, pe găsirea *timpului de execuţie în cazul cel mai defavorabil*, cu alte cuvinte, a celui mai mare timp de execuţie posibil relativ la *orice* date de intrare de dimensiune constanta n. Precizăm trei motive pentru aceasta orientare:

- Timpul de execuţie al unui algoritm în cazul cel mai defavorabil este o margine superioară a timpului de execuţie pentru orice date de intrare de dimensiune fixă. Cunoscând acest timp, avem o garanţie că algoritmul nu va avea, niciodată, un timp de execuţie mai mare. Nu va fi nevoie să facem

presupuneri sau investigaţii suplimentare asupra timpului de execuţie şi să sperăm că acesta nu va fi, niciodată, mult mai mare.

- Pentru anumiţi algoritmi, cazul cel mai defavorabil apare destul de frecvent. De exemplu, în căutarea unei anumite informaţii într-o baza de date, cazul cel mai defavorabil al algoritmului de căutare va apare deseori când informaţia căutată nu este de fapt prezentă în baza de date. În anumite aplicaţii, căutarea unor informaţii absente poate fi frecventă.
- "Cazul mediu" este, adesea, aproape la fel de defavorabil ca şi cazul cel mai defavorabil. Să presupunem că alegem la întâmplare n numere şi aplicam sortarea prin inserţie. Cât timp va fi necesar pentru a determina locul în care putem insera $A[j]$ în subvectorul $A[1..j-1]$? În medie, jumătate din elementele subvectorului $A[1..j-1]$ sunt mai mici decât $A[j]$, şi cealaltă jumătate sunt mai mari. Prin urmare, în medie, trebuie verificate jumătate din elementele subvectorului $A[1..j-1]$, deci $t_j = j/2$. Daca ţinem seama de aceasta observaţie, timpul de execuţie mediu va apărea tot ca o funcţie pătratică de n, la fel ca în cazul cel mai defavorabil.

În anumite cazuri particulare, vom fi interesaţi de *timpul mediu de execuţie* al unui algoritm. O problema care apare în analiza cazului mediu este aceea că s-ar putea să nu fie prea clar din ce sunt constituite datele de intrare "medii" pentru o anumita problema. Adesea, vom presupune că toate datele de intrare având o dimensiune dată sunt la fel de probabile. În practica, aceasta presupunere poate fi falsă, dar un algoritm aleator poate, uneori, să o forţeze.

Ordinul de complexitate (sau de creştere)

Pentru a uşura analiza procedurii Sorteaza-Prin-Inserţie, am utilizat mai multe presupuneri simplificatoare. În primul rând, am ignorat costul real al fiecărei instrucţiuni, folosind constantele c_i pentru a reprezenta aceste costuri. Apoi, am observat că, prin aceste constante, obţinem mai multe detalii decât avem nevoie în mod real: timpul de execuţie în cazul cel mai defavorabil este de forma an^2+bn+c pentru anumite constante a, b şi c care depind de costurile c_i ale instrucţiunilor. Astfel, am ignorat nu numai costurile reale ale instrucţiunilor, dar şi costurile abstracte c_i.

Vom face acum încă o abstractizare simplificatoare. Ceea ce ne interesează de fapt, este *rata de creştere* sau *ordinul de creştere* a timpului de execuţie. Consideram, prin urmare, doar termenul dominant al formulei (adică an^2) deoarece ceilalţi termeni sunt relativ nesemnificativi pentru valori mari ale lui n. Ignorăm, de asemenea, şi factorul constant c, deoarece, pentru numere foarte mari, factorii constanţi sunt mai puţin semnificativi decât rata de creştere în determinarea eficienţei computaţionale a unor algoritmi. Astfel, vom spune, de exemplu, că sortarea prin inserţie are un timp de execuţie în cazul cel mai defavorabil de $\Theta(n^2)$ (pronunţat "teta de *n* pătrat").

În mod uzual, vom considera un algoritm ca fiind mai eficient decât altul dacă timpul său de execuţie în cazul cel mai defavorabil are un ordin de creştere mai mic. Aceasta evaluare ar putea fi incorecta pentru date de intrare de dimensiune mica, dar în cazul unor date de intrare de dimensiuni foarte mari, un algoritm de tipul $\Theta(n^2)$, de exemplu, va fi executat în cazul cel mai defavorabil mult mai repede decât unul de tipul $\Theta(n^3)$.

Aşadar, sortarea prin inserţie are un ordin de complexitate $\Theta(n^2)$.

2. Metoda bulelor (bubble-sort)

Algoritmul constă în parcurgerea tabloului *A* de mai multe ori, până când devine ordonat. La fiecare pas se compară două elemente alăturate. Dacă $a_i > a_{i+1}$, ($i = 1, 2, ..., n - 1$), atunci cele două valori se interschimbă între ele. Controlul acţiunii repetitive este dat de variabila booleană *ok*, care la fiecare reluare a algoritmului primeşte valoarea iniţială *adevărat*, care se schimbă în *fals* dacă s-a efectuat o interschimbare de două elemente alăturate. În momentul în care tabloul *A* s-a parcurs fără să se mai efectueze nici o schimbare, *ok* rămâne cu valoarea iniţială *adevărat* şi algoritmul se termină, deoarece tabloul este ordonat.

Interschimbarea a două elemente se realizează prin intermediul variabilei auxiliare *aux* care are acelaşi tip ca şi elementele tabloului.

```
Subalgoritm  Metoda_bulelor (A)
1: repeta
2:     ok ← adevărat
3:     pentru i=1,n-1 execută
```

```
4:          dacă a[i] > a[i+1] atunci
5:                ok ← fals
6:                aux ← a[i]
7:                a[i] ← a[i+1]
8:                a[i+1] ← aux
9: pana cand ok
```

Considerăm tabloul *A* cu 5 elemente numere reale: 0.0, 1.1, 1.0, 1.2 şi 0.08. Prima parcurgere a tabloului (*ok* este iniţializat cu *adevărat*):

a_1 = 0.0	a_2 = 1.1	a_3 = 1.0	a_4 = 1.2	a_5 = 0.08	*ok*
0.0	1.0 ↔	1.1	1.2	0.08	*fals*
0.0	1.0	1.1	0.08 ↔	1.2	*fals*

Valorile 0.0 < 1.1, rămân neschimbate, 1.1 > 1.0, le interschimbăm. Deoarece 1.1 < 1.2, avansăm şi constatăm că 1.2 > 0.0.8, deci din nou avem interschimbare. În consecinţă, la ieşire din structura pentru *ok* este *fals*. Observăm că 1.2 a ajuns pe locul lui definitiv. Urmează a doua parcurgere a tabloului (*ok* primeşte din nou valoarea *adevărat*).

a_1 = 0.0	a_2 = 1.0	a_3 = 1.1	a_4 = 0.08	a_5 = 1.2	*ok*
0.0	1.0	0.08 ↔	1.1	1.2	*fals*

Am avut interschimbare şi de data aceasta, deci ieşim cu *ok* = *fals*. La acest pas 1.1 a ajuns pe locul său definitiv. A treia parcurgere a tabloului începe cu reiniţializarea lui *ok* cu valoarea *adevărat*.

a_1 = 0.0	a_2 = 1.0	a_3 = 0.08	a_4 = 1.1	a_5 = 1.2	*ok*
0.0	0.08 ↔	1.0	1.1	1.2	*fals*

Am interschimbat 0.08 cu 1.0, cel din urmă astfel a ajuns pe locul său în şirul ordonat. A patra parcurgere a tabloului se finalizează cu valoarea *ok* = *adevărat*, deoarece nu am efectuat nici o interschimbare, ceea ce înseamnă că procesul de ordonare s-a încheiat.

a_1 = 0.0	a_2 = 0.08	a_3 = 1.0	a_4 = 1.1	a_5 = 1.2	*ok*
0.0	0.08	1.0	1.1	1.2	*adevărat*

Observaţia cu privire la faptul că la fiecare parcurgere a ajuns cel puţin un element pe locul său definitiv în şirul ordonat poate fi fructificată, deoarece constatăm că astfel, la următorul pas nu mai sunt necesare verificările în care intervine acest element şi cele care se află după el în şir. Rezultă că la fiecare parcurgere am putea micşora cu 1 numărul elementelor verificate. Dar este posibil că la o parcurgere să ajungă mai multe elemente în locul lor definitiv. Rezultă că vom ţine minte indicele ultimului element care a intervenit în interschimbare şi verificările le vom efectua doar până la acest element. Astfel, ajungem la următorul subalgoritm îmbunătăţit ca performanţă:

```
Subalgoritm  Metoda_bulelor (A)
1: k=n {k va fi limita superioara pana unde se interschimba elemente}
2: repeta
3:     ok ← adevărat
4:     pentru i=1,k-1 execută
5:         dacă a[i] > a[i+1] atunci
6:             ok ← fals
7:             aux ← a[i]
8:             a[i] ← a[i+1]
9:             a[i+1] ← aux
10:            j=i
11:    k=j;  {ultimul indice pentru care s-a facut interchimbare}
12: pana cand ok
```

Metoda bulelor nu este cea mai performantă modalitate de a ordona un şir cu multe elemente, dar în cazul şirurilor „aproape ordonate”, cu optimizările de mai sus, poate deveni mai eficientă decât alte metode.

Pentru a analiza timpul de execuţie al metodei bulelor trebuie să luăm în considerare 3 mărimi:

1. numărul de treceri (parcurgeri)

2. numărul de interschimbări

3. numărul de comparaţii

Dacă presupunem că valorile de intrare reprezintă o permutare a mulţimii {1, 2, ..., n} putem uşor descrie efectul fiecărei treceri astfel: dacă un element ai nu are nici un precedent mai mare el va rămâne pe loc, iar daca are cel puţin unul mai

mare atunci el se va muta cu o poziţie mai în faţă. Reamintesc că elementul maxim de la fiecare parcurgere va ajunge pe poziţia finală. Astfel, eficienţa metodei bulelor depinde de numărul de inversiuni din permutarea iniţială (faţă de permutarea finală, cea identică). Voi defini în continuare inversiunea şi proprietăţile inversiunilor.

Definiţie: Fie a_1, a_2, ... a_n o permutare a mulţimii {1, 2, ..., n}. Dacă i<j şi $a_i > a_j$, atunci perechea (a_i, a_j) se numeşte inversiunii a permutării. De exemplu, în permutarea 3,1,4,2 se găsesc 3 inversiuni: (3,1), (3,2) şi (4,2).

Observaţie: Singura permutare fără inversiuni este cea ordonată (identică) 1,2,...,n.

Cazul cel mai favorabil este acela în care datele iniţiale sunt ordonate crescător, caz în care se face o singură parcurgerea a datelor.

Cazul cel mai nefavorabil este cel în care datele sunt sortate descrescător, caz în care se vor face n-1 parcurgeri. La prima parcurgere se vor face n-1 interschimbări, la a doua n-2 şi aşa mai departe. Aşadar numărul de comparaţii şi cel de interschimbări va fi n(n-1)/2. Exemplu în acest caz:

Numărul parcurgerii	a_1	a_2	a_3	a_4	Numărul de interschimbări
iniţial	4	3	2	1	
1	3	2	1	4	3
2	2	1	3	4	2
3	1	2	3	4	1
Total interschimbări					6

Astfel, vom spune că metoda bulelor are un timp de execuţie în cazul cel mai defavorabil de $\Theta(n^2)$.

3. Sortare stabilind poziţia definitivă prin numărare

Această metodă constă în construirea unui nou tablou *B* care are aceeaşi dimensiune ca şi tabloul *A* în care depunem elementele din *A*, ordonate crescător.

Vom lua fiecare element şi îl vom compara cu fiecare alt element din şir pentru a putea reţine în variabila *k* numărul elementelor care sunt mai mici decât elementul considerat. Astfel, vom afla poziţia pe care trebuie să-l punem pe acesta

în şirul *B*. Dacă în problemă avem nevoie de şirul ordonat tot în tabloul *A*, vom copia în *A* întreg tabloul *B*.

```
Subalgoritm Numărare(n,A)
1: pentru i=1,n execută
2:     k ← 0
3:     pentru j=1,n execută
4:         dacă (A[i] > A[j]) atunci
5:             k ← k + 1          { numărăm câte elemente sunt mai mici decât A[i] }
6:     B[k+1] ← A[i]          { pe A[i] îl punem pe poziţia corespunzătoare din B }
7: A ← B          { copiem peste şirul A întreg şirul B }
```

Exemplu

Fie tabloul *A* cu 4 elemente: 7, 2, 3, –1.

i	*j*	*Relaţia*	*k*	b_{k+1}
1	1	$i = j$	0	
1	2	$7 > 2$	1	
1	3	$7 > 3$	2	
1	4	$7 > -1$	3	$b_4 \leftarrow 7$
2	1	$2 < 7$	0	
2	2	$i = j$	0	
2	3	$2 < 3$	0	
2	4	$2 > -1$	1	$b_2 \leftarrow 2$
3	1	$3 < 7$	0	
3	2	$3 > 2$	1	
3	3	$i = j$	1	
3	4	$3 > -1$	2	$b_3 \leftarrow 3$
4	1	$-1 < 7$	0	
4	2	$-1 < 2$	0	
4	3	$-1 < 3$	0	
4	4	$i = j$	0	$b_1 \leftarrow -1$

Algoritmul funcţionează în această formă dacă elementele tabloului A sunt distincte, altfel în B vor rămâne elemente necopiate din A, deoarece două elemente egale se vor copia în B pe aceeaşi poziţie. De exemplu, pentru un tabloul A cu 4

elemente: 7, 2, 3, 3 se va obţine 2, 3, 0, 7 dacă în B am avut iniţial doar elemente nule.

Această situaţie se poate remedia în 2 moduri:

1. mai parcurgem o dată şirul final şi toate elementele care nu respectă relaţia de ordine le modificăm atribuindu-le valoarea din stânga. Astfel, din şirul 2, 3, 0, 7 vom obţine 2, 3, 3, 7.

Algoritmul modificat este următorul:

```
Subalgoritm Numărare(n,a)
1: pentru i=1,n execută
2:    k ← 0
3:    pentru j=1,n execută
4:       dacă (a[i] > a[j]) atunci
5:          k ← k + 1
6:    b[k+1] ← a[i]
7: a ← b
8: pentru i=1,n-1 execută
9:     dacă  a[i] ≥ a[i+1] atunci
10:       a[i+1] ← a[i]
```

2. înainte de atribuirea b[k+1] ← a[i] verificăm dacă b[k+1] ≠ a[i] şi cât timp e fals mărim pe k. Rezultatul este acelaşi.

Algoritmul modificat este următorul:

```
Subalgoritm Numărare(n,a)
1: pentru i=1,n execută
2:       k ← 0
3:       pentru j=1,n execută
4:          dacă (a[i] > a[j]) şi (i ≠ j) atunci
5:             k ← k + 1
6:       câttimp  b[k+1] = a[i] execută  {cat timp în B a fost deja pus un element cu aceeasi valoare}
8:             k ← k + 1
9:      b[k+1] ← a[i]
10: a ← b
```

Pentru acest algoritm de sortare nu putem spune că există caz favorabil, nefavorabil sau mediu, deoarece numărul de paşi efectuaţi este n^2 indiferent de structura datelor de intrare. Aşadar, ordinul de complexitate este $\Theta(n^2)$.

4. Sortare prin selecţie directă

Metoda precedentă are dezavantajul că necesită de două ori mai multă memorie decât tabloul *A*. Dacă dorim să evităm această risipă, putem aplica metoda de ordonare prin selectarea unui element şi plasarea lui pe poziţia sa finală direct în tabloul A.

De exemplu, în caz de ordonare crescătoare, pornind de la primul element se caută valoarea minimă din tablou. Aceasta se aşează pe prima poziţie printr-o interschimbare între elementul de pe prima poziţie şi elementul minim. Apoi, se reia algoritmul, pornind de la a doua poziţie şi se caută minimul între elementele a_2, ..., a_n. Acesta se interschimbă cu al doilea dacă este cazul. Procedeul se continuă până la ultimul element.

Pseudocodul algoritmului de sortare prin selecţia minimului este:

```
Subalgoritm Selecţie(n,a)
1:  pentru i=1,n-1 execută:
2:      min ← a[i]
3:      pentru j=i+1,n execută:
4:          dacă min > a[j] atunci
5:              min ← a[j]
6:              k ← j
7:      dacă min ≠ a[i] atunci
8:          aux ← a[i]
9:          a[i] ← a[k]
10:         a[k] ← aux
```

Exemplu

Fie tabloul *A* = (5, 0, 8, 7, 3).

Pas	*Tabloul A*	*Element minim*	*Poziţia minimului*	*Noul tablou A*
1	(5, **0**, 8, 7, 3)	0	2	(**0**, 5, 8, 7, 3)
2	(0, 5, 8, 7, **3**)	3	5	(0, **3**, 8, 7, 5)
3	(0, 3, 8, 7, **5**)	5	5	(0, 3, **5**, 7, 8)
4	(0, 3, 5, **7**, 8)	7	4	

Algoritmul se poate scrie şi prin determinarea valorilor maxime şi mutarea lor în tablou de la dreapta la stânga, astfel rezultând, de asemenea, un şir ordonat

crescător. Un astfel de algoritm este des utilizat pentru că este mai simplu de reţinut, în forma următoare:

```
Subalgoritm Selecţie(n,a)
1:  pentru i=1,n-1 execută:
2:      pentru j=i+1,n execută:
3:          dacă a[i] > a[j] atunci
4:                  aux ← a[i]
5:                  a[i] ← a[j]
6:                  a[j] ← aux
```

Pentru acest algoritm de sortare nu putem spune că există caz favorabil, nefavorabil sau mediu, deoarece numărul de paşi efectuaţi este n(n-1)/2 indiferent de structura datelor de intrare. Aşadar, ordinul de complexitate este $\Theta(n^2)$. Numărul de comparări este tot n(n-1)/2, însă pentru cazul în care şirul este ordonat crescător se vor face doar n atribuiri (se execută doar instrucţiunea de atribuire 2:)

5. Sortarea rapidă (quick-sort)

Sortarea rapidă este un algoritm de sortare care, pentru un sir de n elemente, are un timp de execuţie $\Theta(n^2)$, în cazul cel mai defavorabil. În ciuda acestei comportări proaste, în cazul cel mai defavorabil, algoritmul de sortare rapidă este deseori cea mai bună soluţie practică, deoarece are o comportare medie remarcabilă: timpul său mediu de execuţie este $\Theta(n \log_2 n)$, şi constanta ascunsă în formula $\Theta(n \log_2 n)$ este destul de mica. Algoritmul are avantajul că sortează pe loc (în spaţiul alocat şirului de intrare) şi lucrează foarte bine chiar şi într-un mediu de memorie virtuală.

Algoritmul conţine şi un subalgoritm important, folosit de sortarea rapidă pentru partiţionare.

Algoritmul de sortare rapidă, ca de altfel şi algoritmul de sortare prin interclasare, se bazează pe paradigma “divide şi stăpâneşte”. Iată un proces “divide şi stăpâneşte” în trei paşi, pentru un subşir $A[p \ldots r]$.

Divide: Şirul $A[p..r]$ este împărţit (rearanjat) în doua subşiruri nevide $A[p..q]$ şi $A[q + 1..r]$, astfel încât fiecare element al subşirului $A[p..q]$ să fie mai mic

sau egal cu orice element al subşirului $A[q + 1..r]$. Indicele q este calculat de procedura de partiţionare.

Stăpâneşte: Cele două subşiruri $A[p..q]$ şi $A[q + 1..r]$ sunt sortate prin apeluri recursive ale algoritmului de sortare rapidă.

Combina: Deoarece cele două subşiruri sunt sortate pe loc, nu este nevoie de nici o combinare, şirul $A[p..r]$ este ordonat.

Descrierea algoritmului este următoarea:

```
Subalgoritm Quicksort(A, p, r)
1: daca p < r atunci
2:     q ← Partiţie(A, p, r)
3:     Quicksort(A, p, q)
4:     Quicksort(A, q + 1, r)
```

Pentru ordonarea întregului sir A, iniţial se apelează Quicksort(A, 1, *lungime*[A]).

Cheia algoritmului este procedura Partiţie, care rearanjează pe loc subşirul $A[p..r]$.

```
Subalgoritm Partiţie(A, p, r)
1: x ← A[p]
2: i ← p - 1
3: j ← r + 1
4: cât timp i<j executa
5:     repeta
6:         j ← j - 1
7:     pana când A[j] ≤ x
8:     repeta
9:         i ← i + 1
10:    pana când A[i] ≥ x
11:    daca i < j atunci
12:        interschimba A[i] ↔ A[j]
13: returneaza j
```

În figura 1.3 este ilustrat modul de funcţionare a procedurii Partiţie. Întâi se selectează un element $x = A[p]$ din şirul $A[p..r]$, care va fi elementul "pivot", în jurul căruia se face partiţionarea şirului $A[p..r]$. Apoi, doua subşiruri $A[p..i]$ şi $A[j..r]$ cresc la începutul şi respectiv, sfârşitul şirului $A[p..r]$, astfel încât fiecare element al şirului $A[p..i]$ să fie mai mic sau egal cu x, şi orice element al şirului $A[j..r]$, mai

mare sau egal cu x. La început $i = p - 1$ şi $j = r + 1$, deci cele două subşiruri sunt vide.

În interiorul ciclului cât timp, în liniile 5–7, indicele j se decrementează, iar se incrementează până când $A[i] \geq x \geq A[j]$. Presupunând că inegalităţile de mai sus sunt stricte, $A[i]$ este prea mare ca să aparţină primului subşir (cel de la început), iar $A[j]$ prea mic ca să aparţină celui de al doilea subşir (cel de la sfârşit). Astfel, interschimbând $A[i]$ cu $A[j]$ (linia 12), cele doua parţi cresc. (Interschimbarea se poate face şi în cazul în care avem inegalităţi stricte.) Ciclul cât timp se repeta până când inegalitatea $i \geq j$ devine adevărata. În acest moment, întregul sir $A[p..r]$ este partiţionat în doua subşiruri $A[p..q]$ şi $A[q + 1..r]$, astfel încât $p \leq q < r$ şi nici un element din $A[p..q]$ nu este mai mare decât orice element din $A[q + 1..r]$. Procedura returnează valoarea $q = j$.

De fapt, procedura de partiţionare executa o operaţie simplă: pune elementele mai mici decât x în primul subşir, iar pe cele mai mari decât x în subşirul al doilea. Exista câteva particularităţi care determina o comportare interesanta a procedurii Partiţie. De exemplu, indicii i şi j nu depăşesc niciodată marginile vectorului $A[p..r]$, dar acest lucru nu se vede imediat din textul procedurii.

Timpul de execuţie al procedurii Partiţie, în cazul unui vector $A[p..r]$, este $\Theta(n)$,

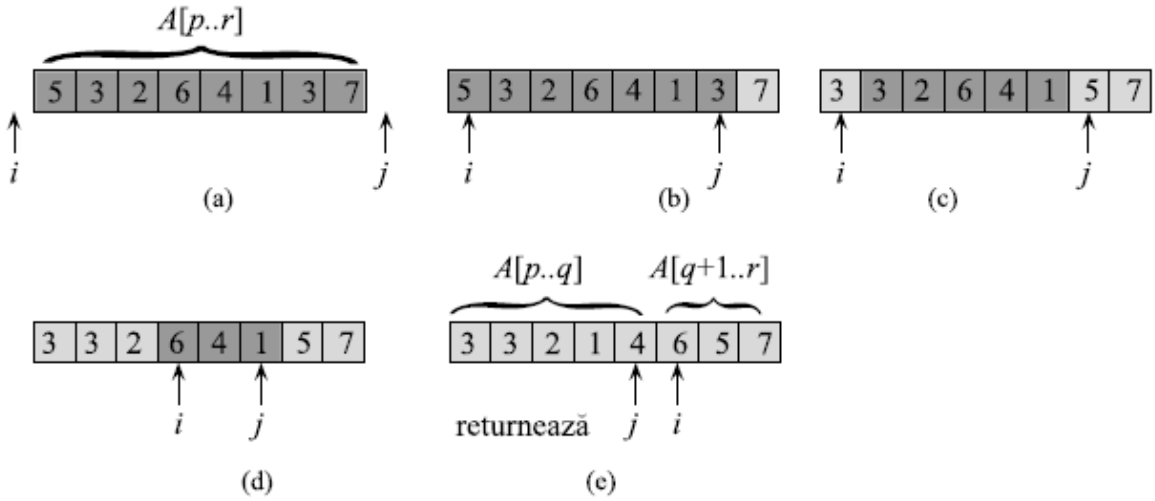

Figura 1.3 Operaţiile efectuate de procedura Partiţie pe un exemplu. Elementele haşurate în gri deschis sunt deja plasate în poziţiile lor corecte, iar cele haşurate închis încă nu. (a) Şirul de intrare, cu valorile iniţiale ale variabilelor i şi j, care punctează în afara şirului. Vom face partiţionarea în jurul elementului $x = A[p] = 5$. (b) Poziţiile lui i şi j în linia 11 a algoritmului, dupa prima parcurgere a ciclului cât timp. (c) Rezultatul schimbului de elemente descris în linia 12. (d) Valorile lui i şi j în linia 11 dupa a doua

parcurgere a ciclului cât timp. (e) Valorile lui *i* şi *j* dupa a treia şi ultima iteraţie a ciclului cât timp. Procedura se termina deoarece *i* ¸ *j* şi valoarea returnata este *q* = *j*. Elementele şirului până la *A*[*j*], inclusiv, sunt mai mici sau egale cu *x* = 5, iar cele de dupa *A*[*j*], sunt toate mai mari sau egale cu *x* = 5.

Timpul de execuţie al algoritmului de sortare rapidă depinde de faptul că partiţionarea este echilibrata sau nu, iar acesta din urmă de elementele alese ca „pivot" pentru partiţionare. Daca partiţionarea este echilibrata, algoritmul este la fel de rapid ca sortarea prin interclasare. În cazul în care partiţionarea nu este echilibrată, algoritmul se executa la fel de încet ca sortarea prin inserare. În aceasta secţiune vom investiga, fără rigoare matematica, performanţa algoritmului de sortare rapidă în cazul partiţionării echilibrate.

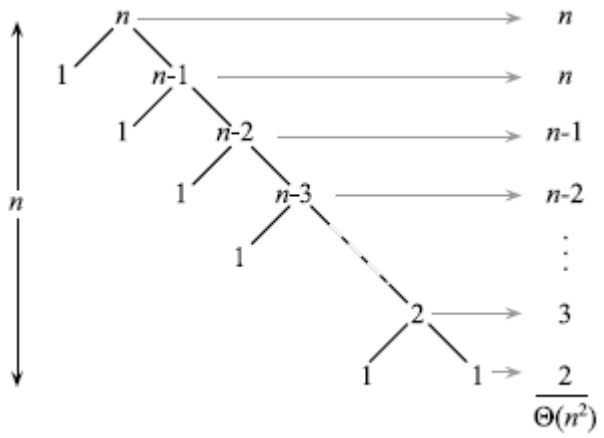

Figura 1.4 Arborele de recursivitate pentru Quicksort când procedura Partiţie pune întotdeauna într-o parte a vectorului numai un singur element (cazul cel mai defavorabil). Timpul de execuţie în acest caz este $\Theta(n^2)$.

Partiţionarea în cazul cel mai defavorabil

Comportarea cea mai defavorabilă a algoritmului de sortare rapidă apare în situaţia în care procedura de partiţionare produce un vector de *n*-1 elemente şi unul de 1 element.

Să presupunem că această partiţionare dezechilibrată apare la fiecare pas al algoritmului. Deoarece timpul de partiţionare este de $\Theta(n)$, şi $T(1) = \Theta(1)$, formula recursiva pentru timpul de execuţie a algoritmului de sortare rapidă este:

$T(n) = T(n - 1) + \Theta(n)$:

Pentru evaluarea formulei de mai sus, observam că $T(1) = \Theta(1)$, apoi iterăm formula:

$$T(n) = T(n-1) + \Theta(n) = \sum_{k=1}^{n} \Theta(k) = \Theta\left(\sum_{k=1}^{n} k\right) = \Theta(n^2).$$

Ultima egalitate se obţine din observaţia că ultima sumă este o progresie aritmetică.

În figura 1.4 este ilustrat arborele de recursivitate pentru acest cel mai defavorabil caz al algoritmului de sortare rapidă. Daca partiţionarea este total dezechilibrata la fiecare pas recursiv al algoritmului, atunci timpul de execuţie este $\Theta(n^2)$. Deci timpul de execuţie, în cazul cel mai defavorabil, nu este mai bun decât al algoritmului de sortare prin inserare, de exemplu. Mai mult, timpul de execuţie este $\Theta(n^2)$ chiar şi în cazul în care vectorul de intrare este ordonat – caz în care algoritmul de sortare prin inserare are timpul de execuţie $\Theta(n)$.

Partiţionarea în cazul cel mai favorabil

Daca algoritmul de partiţionare produce doi vectori de $n/2$ elemente, algoritmul de sortare rapidă lucrează mult mai repede.

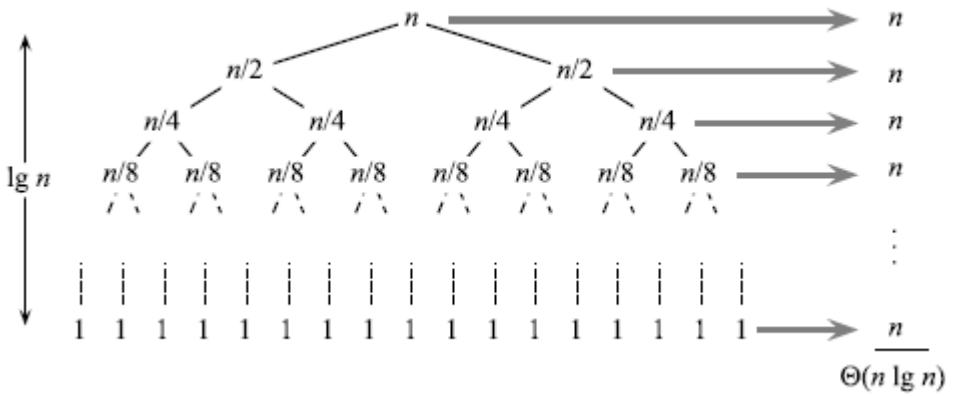

Figura 1.5 Arborele de recurenţă pentru Quicksort când procedura Partiţie produce întotdeauna parţi egale (cazul cel mai favorabil). Timpul de execuţie rezultat este $\Theta(n \log_2 n)$.

Formula de recurenţa în acest caz este: $T(n) = 2T(n/2) + \Theta(n)$, iar soluţia este $T(n) = \Theta(n \log 2\ n)$. Deci partiţionarea cea mai buna produce un algoritm de sortare mult mai rapid. În figura 1.5 se ilustrează arborele de recursivitate pentru acest cel mai favorabil caz. Partiţionarea echilibrată arată că timpul mediu de execuţie a algoritmului de sortare rapidă este mult mai apropiat de timpul cel mai bun decât de timpul cel mai rău. Pentru a înţelege de ce este aşa, ar trebui să studiem efectul partiţionării echilibrate asupra formulei recursive care descrie timpul de execuţie.

Să presupunem că procedura de partiţionare produce întotdeauna o împărţire în proporţie de 9 la 1, care la prima vedere pare a fi o partiţionare dezechilibrata. În acest caz, formula recursivă pentru timpul de execuţie al algoritmului de sortare rapidă este:

$T(n) = T(9n/10) + T(n/10) + n$

unde, pentru simplificare, în loc de $\Theta(n)$ s-a pus n. Arborele de recurenţa corespunzător se găseşte în figura 1.6. Să observam că la fiecare nivel al arborelui costul este n până când la adâncimea $\log_{10} n = \Theta(\log_2 n)$ se atinge o condiţie iniţială. În continuare, la celelalte niveluri, costul nu depăşeşte valoarea n. Apelul recursiv se termina la adâncimea $\log_{10/9} n = \Theta(\log_2 n)$.

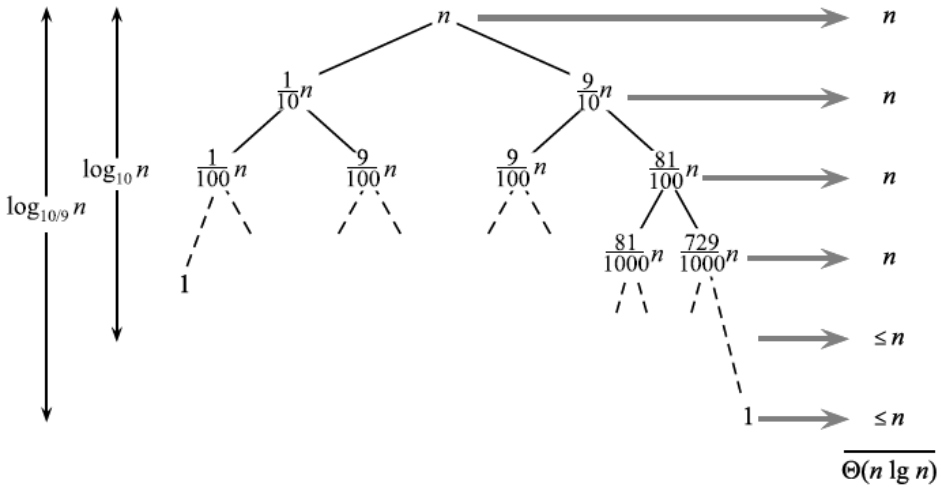

Figura 1.6 Arborele de recurenţa pentru Quicksort, când procedura Partiţie produce întotdeauna parţi în proporţie de 9 la 1, rezultând un timp de execuţie de $\Theta(n \log_2 n)$.

Costul total al algoritmului de sortare rapidă este deci $\Theta(n \log_2 n)$. Ca urmare, cu o partiţionare în proporţie de 9 la 1 la fiecare nivel al partiţionării (care intuitiv pare a fi total dezechilibrata), algoritmul de sortare rapidă are un timp de execuţie de $\Theta(n \log_2 n)$ – asimptotic acelaşi ca în cazul partiţionării în două parţi egale. De fapt, timpul de execuţie va fi $\Theta(n \log_2 n)$ şi în cazul partiţionării într-o proporţie de 99 la 1. La orice partiţionare într-o proporţie *constanta*, adâncimea arborelui de recursivitate este $\Theta(\log_2 n)$ şi costul, la fiecare nivel, este $\Theta(n)$. Deci timpul de execuţie este $\Theta(n \log_2 n)$ la orice partiţionare într-o proporţie constantă.

Intuirea comportării medii

Pentru a avea o idee clara asupra comportării medii a algoritmului de sortare rapidă, trebuie să facem presupuneri asupra frecvenţei anumitor intrări. Cea mai evidenta presupunere este că toate permutările elementelor de intrare sunt la fel de probabile. Vom discuta aceasta presupunere în secţiunea următoare, aici vom exploata doar câteva variante.

În situaţia în care algoritmul de sortare rapidă lucrează pe o intrare aleatoare, probabil că nu va partiţiona la fel la fiecare nivel, cum am presupus în discuţiile anterioare. Este de aşteptat ca unele partiţionări să fie echilibrate, altele nu.

În cazul mediu, procedura Partiţie produce un amestec de partiţionări "bune" şi "rele". Într-un arbore de recurenţa pentru cazul mediu al procedurii Partiţie, partiţionările bune şi rele sunt distribuite aleator. Să presupunem, totuşi, pentru simplificare, că partiţionările bune şi rele alternează pe niveluri, şi că partiţionările bune corespund celui mai bun caz, iar cele rele celui mai defavorabil caz. În figura 1.7 sunt prezentate partiţionările la doua niveluri consecutive în arborele de recursivitate. Costul partiţionării la rădăcina arborelui este n, iar vectorii obţinuţi sunt de dimensiune n - 1 şi 1: cazul cel mai defavorabil. La nivelul următor, vectorul de n - 1 elemente se împarte în doi vectori de (n - 1)/2 elemente fiecare, potrivit cazului celui mai bun.

Să presupunem că pentru un vector de dimensiune 1 (un element) costul este 1.

Combinarea unei partiţionări rele şi a uneia bune produce trei vectori de dimensiune 1, (n - 1)/2 şi respectiv (n - 1)/2, cu un cost total de $2n - 1 = \Theta(n)$. Evident, aceasta situaţie nu este mai rea decât cea prezentata în figura 1.7 (b), adică cea cu un singur nivel, care produce un vector de (n - 1)/2 + 1 elemente şi unul de (n - 1)/2 elemente, cu un cost total de $n = \Theta(n)$.

Totuşi, situaţia din urmă este aproape echilibrată, cu siguranţă mult mai bună decât proporţia 9 la 1. Intuitiv, o partiţionare defavorabila de un cost $\Theta(n)$ poate fi absorbită de una buna tot de un cost $\Theta(n)$, şi partiţionarea rezultata este favorabila. Astfel timpul de execuţie al algoritmului de sortare rapidă, când

partiţionările bune şi rele alternează, este acelaşi ca în cazul partiţionărilor bune: tot Θ ($n \log_2 n$), doar constanta din notaţia Θ este mai mare.

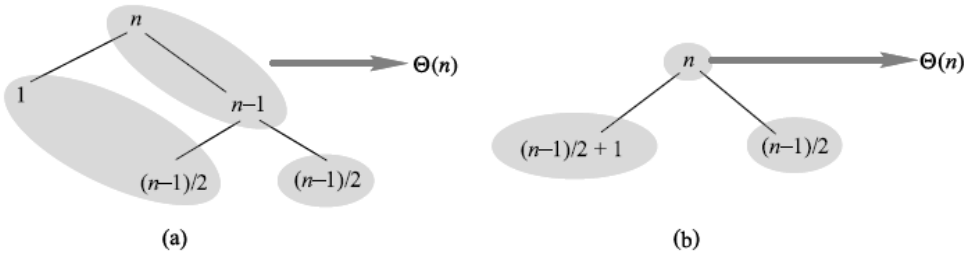

Figura 1.7 (a) Două niveluri ale arborelui de recurenţă pentru algoritmul de sortare rapidă. Partiţionarea la nivelul rădăcinii consumă *n* unităţi de timp şi produce o partiţionare "proastă": doi vectori de dimensiune 1 şi *n* - 1. Partiţionarea unui subşir de *n* - 1 elemente necesita *n* - 1 unităţi de timp şi este o partiţionare "bună": produce doua subşiruri de (*n* - 1)/2 elemente fiecare. (b) Un singur nivel al arborelui de recurenţa care este mai rău decât nivelurile combinate de la (a), totuşi foarte bine echilibrat.

6. Heapsort

Prin algoritmul heapsort se ordonează elementele în spaţiul alocat vectorului: la un moment dat doar un număr constant de elemente ale vectorului sunt păstrate în afara spaţiului alocat vectorului de intrare. Astfel, algoritmul heapsort combină calităţile a două tipuri de algoritmi de sortare, sortare internă şi sortare externă.

Heapsort introduce o tehnica nouă de proiectare a algoritmilor bazată pe utilizarea unei structuri de date, numită de regula *heap*. Structura de date heap este utilă nu doar pentru algoritmul heapsort, ea poate fi la fel de utilă şi în tratarea eficientă a unei cozi de prioritate.

Termenul heap a fost introdus şi utilizat iniţial în contextul algoritmului heapsort, dar acesta se foloseşte şi în legătură cu alocarea dinamică, respectiv în tratarea memoriei bazate pe "colectarea reziduurilor" (*garbage collected storage*), de exemplu în limbajele de tip Lisp.

Structura de date heap *nu* se refera la heap-ul menţionat în alocarea dinamica, şi ori de câte ori, în aceasta lucrare voi vorbi despre heap, vom înţelege structura definită aici pentru heapsort.

Structura de date *heap (binar)* este un vector care poate fi vizualizat sub forma unui arbore binar aproape complet, conform figurii 1.8. Fiecare nod al arborelui corespunde unui element al vectorului care conţine valorile ataşate

nodurilor. Arborele este plin, exceptând eventual nivelul inferior, care este plin de la stânga la dreapta doar până la un anumit loc. Un vector *A* care reprezintă un heap are doua atribute: *lungime*[*A*], reprezintă numărul elementelor din vector şi *dimensiune-heap*[*A*] reprezintă numărul elementelor heap-ului memorat în vectorul *A*. Astfel, chiar daca *A*[1..*lungime*[*A*]] conţine în fiecare element al său date valide, este posibil ca elementele următoare elementului *A*[*dimensiune-heap*[*A*]], unde *dimensiune- heap*[*A*] ≤ *lungime*[*A*], să nu aparţină heap-ului. Rădăcina arborelui este *A*[1]. Dat fiind un indice *i*, corespunzător unui nod, se pot determina uşor indicii părintelui acestuia Parinte(*i*), al fiului Stânga(*i*) şi al fiului Dreapta(*i*).

```
Parinte(i)
    returneaza [i/2]
Stânga(i)
    returneaza 2i
Dreapta(i)
    returneaza 2i + 1
```

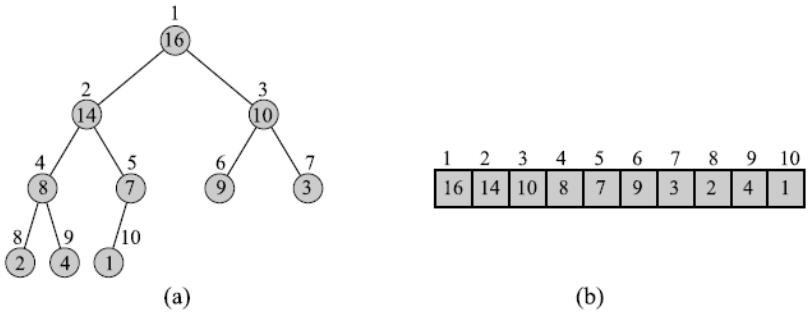

Figura 1.8 Un heap reprezentat sub forma unui arbore binar (a) şi sub forma unui vector (b). Numerele înscrise în cercurile reprezentând nodurile arborelui sunt valorile ataşate nodurilor, iar cele scrise lângă cercuri sunt indicii elementelor corespunzătoare din vector.

În cele mai multe cazuri, procedura Stânga poate calcula valoarea 2*i* cu o singura instrucţiune, translatând reprezentarea binara a lui *i* la stânga cu o poziţie binara. Similar, procedura Dreapta poate determina rapid valoarea 2*i* + 1, translatând reprezentarea binara a lui *i* la stânga cu o poziţie binara, iar bitul nou intrat pe poziţia binara cea mai nesemnificativa va fi 1. În procedura Parinte valoarea [*i*/2] se va calcula prin translatarea cu o poziţie binara la dreapta a reprezentării binare a lui *i*. Într-o implementare eficienta a algoritmului heapsort, aceste proceduri sunt adeseori codificate sub forma unor “macro-uri” sau a unor proceduri “în-line”.

Pentru orice nod *i*, diferit de rădăcina, este adevărată următoarea *proprietate de heap*:

$A[\text{Parinte}(i)] \geq A[i]$

adică valoarea ataşata nodului este mai mică sau egală cu valoarea asociată părintelui său. Astfel cel mai mare element din heap este păstrat în rădăcină, iar valorile nodurilor oricărui subarbore al unui nod sunt mai mici sau egale cu valoarea nodului respectiv.

Definim *înălţimea* unui nod al arborelui ca fiind numărul muchiilor aparţinând celui mai lung drum care leagă nodul respectiv cu o frunză, iar înălţimea arborelui ca fiind înălţimea rădăcinii. Deoarece un heap având n elemente corespunde unui arbore binar complet, înălţimea acestuia este $\Theta(\log_2 n)$. Vom vedea că timpul de execuţie al operaţiilor de bază, care se efectuează pe un heap, este proporţional cu înălţimea arborelui şi este $\Theta(\log_2 n)$. În cele ce urmează, vom prezenta trei proceduri şi modul lor de utilizare în algoritmul de sortare, respectiv într-o structura de tip coada de prioritate.

- Procedura Reconstituie-Heap are timpul de execuţie $\Theta(\log_2 n)$ şi este de prima importanţa în întreţinerea proprietăţii de heap.
- Procedura Construieste-Heap are un timp de execuţie liniar şi generează un heap dintr-un vector neordonat, furnizat la intrare.
- Procedura Heapsort se executa în timpul $O(n \log_2 n)$ şi ordonează un vector în spaţiul alocat acestuia.

Procedura Reconstituie-Heap este un subprogram important în prelucrarea heap-urilor.

Datele de intrare ale acesteia sunt un vector *A* şi un indice *i* din vector. Atunci când se apelează Reconstituie-Heap, se presupune că subarborii, având ca rădăcini nodurile Stânga(*i*) respectiv Dreapta(*i*), sunt heap-uri. Dar, cum elementul $A[i]$ poate fi mai mic decât descendenţii săi, acesta nu respecta proprietatea de heap. Sarcina procedurii Reconstituie-Heap este de a "scufunda" în heap valoarea $A[i]$, astfel încât subarborele care are în rădăcina valoarea elementului de indice *i*, să devina un heap.

```
Subalgoritm Reconstituie-Heap(A, i)
1: l ← Stânga(i)
2: r ←Dreapta(i)
3: daca l ≤ dimesiune-heap[A] şi A[l] > A[i] atunci
4:      maxim ←l
5: altfel
6:      maxim ←i
7: daca r ≤ dimesiune-heap[A] şi A[r] > A[maxim] atunci
8:      maxim ←r
9: daca maxim ≠ i atunci
10:     schimba A[i] ↔ A[maxim]
11:     Reconstituie-Heap(A, maxim)
```

Figura 1.9 ilustrează efectul procedurii Reconstituie-Heap. La fiecare pas se determina cel mai mare element dintre $A[i]$, A[Stânga(i)] şi A[Dreapta(i)], iar indicele său se păstrează în variabila *maxim*. Daca $A[i]$ este cel mai mare, atunci subarborele având ca rădăcină nodul i este un heap şi procedura se termina. În caz contrar, cel mai mare element este unul dintre cei doi descendenţi şi $A[i]$ este interschimbat cu A[*maxim*]. Astfel, nodul i şi descendenţii săi satisfac proprietatea de heap. Nodul *maxim* are acum valoarea iniţiala a lui $A[i]$, deci este posibil ca subarborele de rădăcină *maxim* să nu îndeplinească proprietatea de heap. Rezulta că procedura Reconstituie-Heap trebuie apelata recursiv din nou pentru acest subarbore.

Timpul de execuţie al procedurii Reconstituie-Heap, corespunzător unui arbore de rădăcină i şi dimensiune n, este $\Theta(1)$, timp în care se pot analiza relaţiile dintre $A[i]$, A[Stânga(i)] şi A[Dreapta(i)] la care trebuie adăugat timpul în care Reconstituie-Heap se executa pentru subarborele având ca rădăcină unul dintre descendenţii lui i. Dimensiunea acestor subarbori este de cel mult $2n/3$ – cazul cel mai defavorabil fiind acela în care nivelul inferior al arborelui este plin exact pe jumătate – astfel, timpul de execuţie al procedurii Reconstituie-Heap poate fi descris prin următoarea inegalitate recursiva:

$T(n) \leq T(2n/3) + \Theta(1)$*:*

Timpul de execuţie al procedurii Reconstituie-Heap pentru un nod de înălţime h poate fi exprimat alternativ ca fiind egal cu $\Theta(h)$.

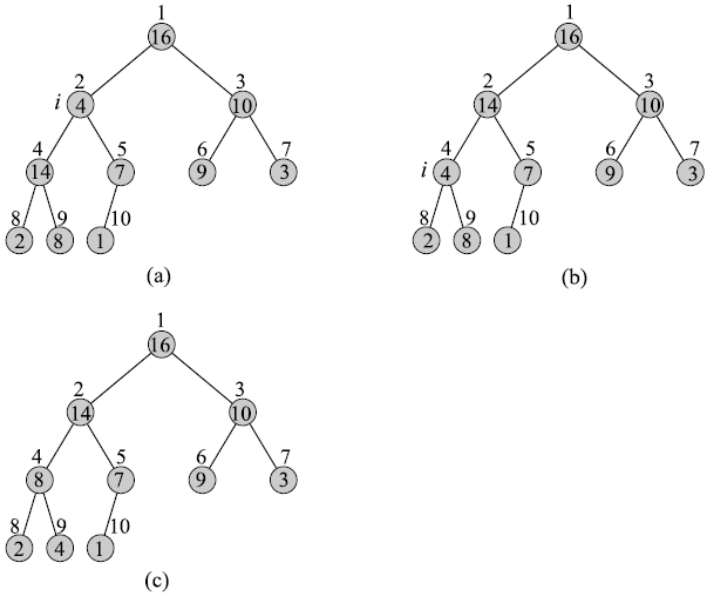

Figura 1.9 Efectul procedurii Reconstituie-Heap(*A*, 2), unde *dimesiune-heap*[*A*] = 10. (a) Configuraţia iniţiala a heap-ului, unde *A*[2] (pentru nodul *i* = 2), nu respecta proprietatea de heap deoarece nu este mai mare decât descendenţii săi. Proprietatea de heap este restabilita pentru nodul 2 în (b) prin interschimbarea lui *A*[2] cu *A*[4], ceea ce anulează proprietatea de heap pentru nodul 4. Apelul recursiv al procedurii Reconstituie-Heap(*A*, 4) poziţionează valoarea lui *i* pe 4. Dupa interschimbarea lui *A*[4] cu *A*[9], aşa cum se vede în (c), nodul 4 ajunge la locul său şi apelul recursiv Reconstituie-Heap(*A*, 9) nu mai găseşte elemente care nu îndeplinesc proprietatea de heap.

Construirea unui heap

Procedura Reconstituie-Heap poate fi utilizata "de jos în sus" pentru transformarea vectorului *A*[1..*n*] în heap, unde *n* = *lungime*[*A*]. Deoarece toate elementele subşirului *A*[([*n*/2]+1)..*n*] sunt frunze, acestea pot fi considerate ca fiind heap-uri formate din câte un element. Astfel, procedura Construieste-Heap trebuie să traverseze doar restul elementelor şi să execute procedura Reconstituie-Heap pentru fiecare nod întâlnit. Ordinea de prelucrare a nodurilor asigura că subarborii, având ca rădăcină descendenţi ai nodului *i* să formeze heap-uri înainte ca Reconstituie-Heap să fie executat pentru aceste noduri.

```
Subalgoritm Construieste-Heap(A)
1: dimesiune-heap[A] ← lungime[A]
2: pentru i ← [lungime[A]/2],1 executa
3:     Reconstituie-Heap(A, i)
```

Figura 1.10 ilustrează modul de funcţionare al procedurii Construieste-Heap.

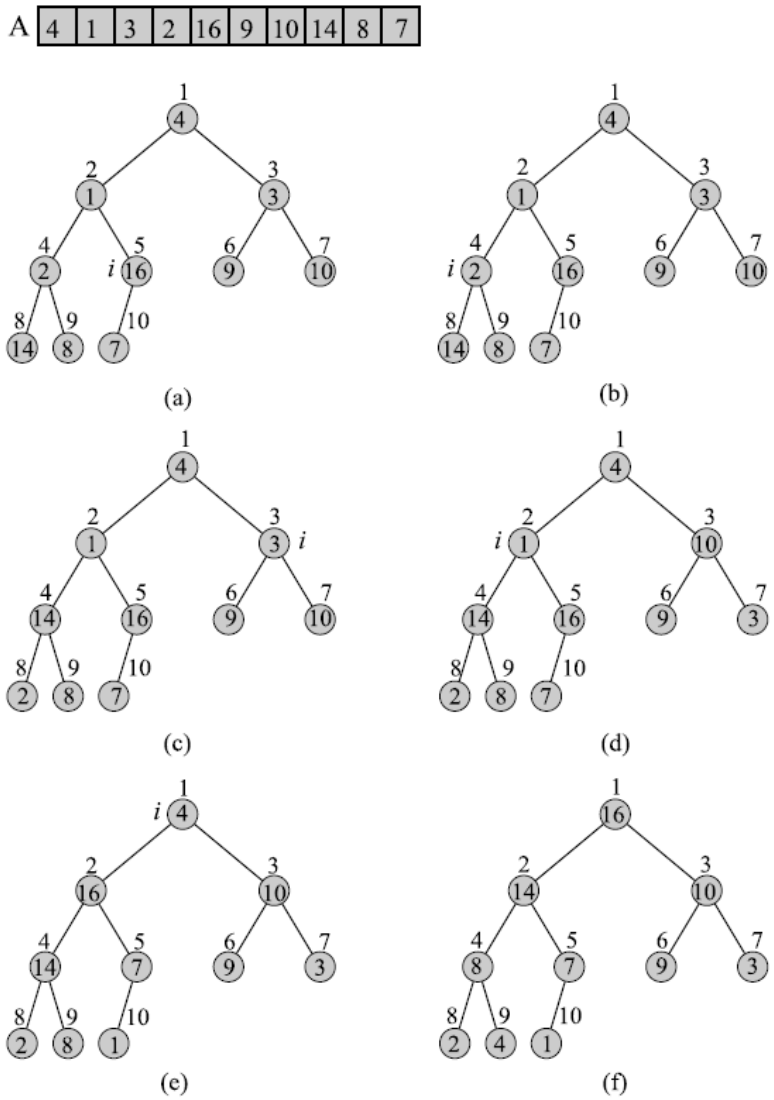

Figura 1.10 Modul de execuţie a procedurii Construieste-Heap. În figura se vizualizează structurile de date în starea lor anterioara apelului procedurii Reconstituie-Heap (linia 3 din procedura Construieste-Heap). (a) Se considera un vector *A* având 10 elemente şi arborele binar corespunzător. Dupa cum se vede în figura, variabila de control *i* a ciclului, în momentul apelului Reconstituie-Heap(*A*, *i*), indica nodul 5. (b) reprezintă rezultatul, variabila de control *i* a ciclului acum indica nodul 4. (c) - (e) vizualizează iteraţiile succesive ale ciclului pentru din Construieste-Heap. Se observa că, atunci când se apelează procedura Reconstituie-Heap pentru un nod dat, subarborii acestui nod sunt deja heap-uri. (f) reprezintă heap-ul final al procedurii Construieste-Heap.

Timpul de execuţie al procedurii Construieste-Heap poate fi calculat simplu, determinând limita superioara a acestuia: fiecare apel al procedurii Reconstituie-Heap necesita un timp $\Theta(\log_2 n)$ şi, deoarece pot fi $\Theta(n)$ asemenea apeluri, rezulta că timpul de execuţie poate fi cel mult $\Theta(n \log_2 n)$. Aceasta estimare este corecta, dar nu este suficient de tare asimptotic.

Vom obţine o limita mai tare observând că timpul de execuţie a procedurii Reconstituie-Heap depinde de înălţimea nodului în arbore, aceasta fiind mica pentru majoritatea nodurilor.

Estimarea noastră mai riguroasa se bazează pe faptul că un heap având n elemente are înălţimea $\log_2 n$ şi că pentru orice înălţime h, în heap exista cel mult $[n/2^{h+1}]$ noduri de înălţime h.

Timpul de execuţie a procedurii Reconstituie-Heap pentru un nod de înălţime h fiind $\Theta(h)$, obţinem pentru timpul de execuţie a procedurii Construieste-Heap:

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right)$$

Astfel, timpul de execuţie al procedurii Construieste-Heap poate fi estimat ca fiind:

$$O\left(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h} \right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(n)$$

De aici rezulta că se poate construi un heap dintr-un vector într-un timp liniar.

Algoritmul heapsort

Algoritmul heapsort începe cu apelul procedurii Construieste-Heap în scopul transformării vectorului de intrare $A[1..n]$ în heap, unde n = *lungime*$[A]$. Deoarece cel mai mare element al vectorului este ataşat nodului rădăcină $A[1]$, acesta va ocupa locul definitiv în vectorul ordonat prin interschimbarea sa cu $A[n]$. În continuare, "excluzând" din heap cel de-al n-lea element (şi micşorând cu 1 *dimesiune-heap*$[A]$), restul de $A[1..(n - 1)]$ elemente se pot transforma uşor în heap, deoarece subarborii nodului rădăcină au proprietatea de heap, cu eventuala excepţie a elementului ajuns în nodul rădăcină.

```
Subalgoritm Heapsort(A)
1: Construieste-Heap(A)
2: pentru i ← lungime[A], 2 executa
3:     schimba A[1] ↔ A[i]
4:     dimesiune-heap[A] ← dimesiune-heap[A] - 1
```

5:	Reconstituie-Heap(*A*, 1)

Apelând procedura Reconstituie-Heap(*A*, 1) se restabileşte proprietatea de heap pentru vectorul *A*[1..(*n* - 1)]. Acest procedeu se repeta micşorând dimensiunea heap-ului de la *n* - 1 până la 2.

Figura 1.11 ilustrează, pe un exemplu, modul de funcţionare a procedurii Heapsort, dupa ce în prealabil datele au fost transformate în heap. Fiecare heap reprezintă starea iniţiala la începutul pasului iterativ (linia 2 din ciclul pentru).

Timpul de execuţie al procedurii Heapsort este $\Theta(n \log_2 n)$, deoarece procedura Construieste-Heap se executa într-un timp $\Theta(n)$, iar procedura Reconstituie-Heap, apelata de n-1 ori, se executa în timpul $\Theta(\log_2 n)$.

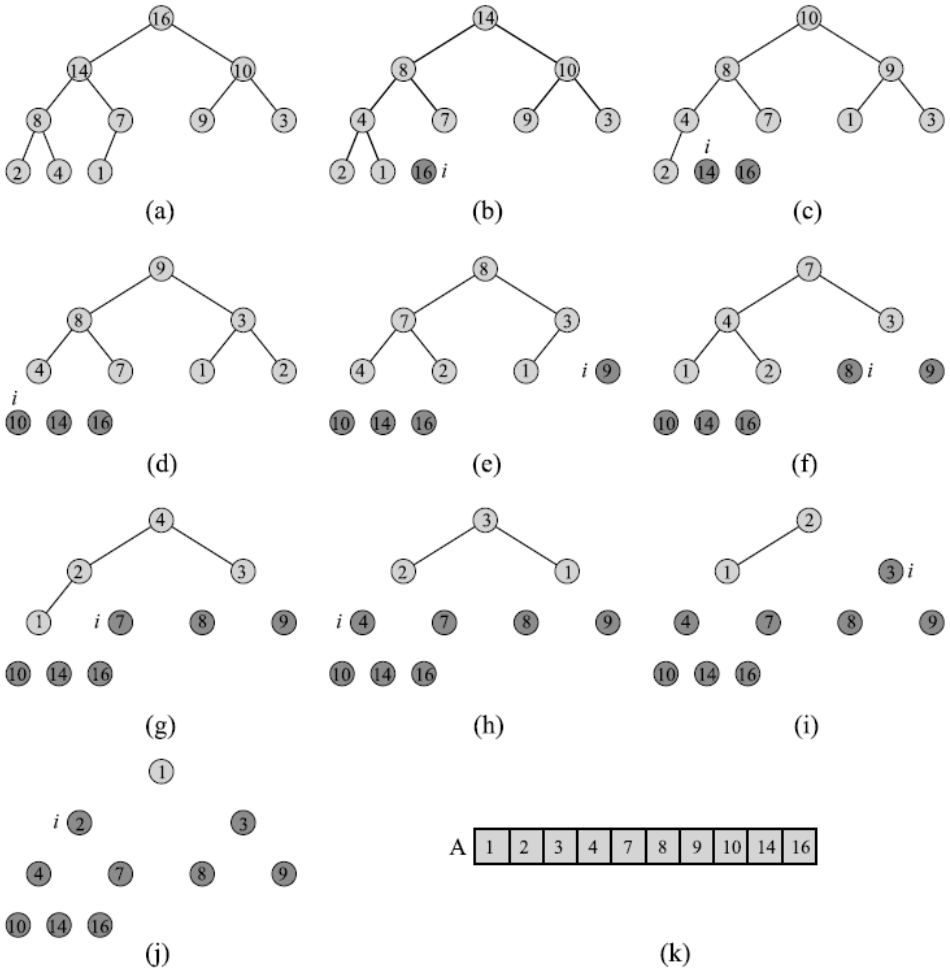

Figura 1.11 Modul de funcţionare a algoritmului Heapsort. (a) Structura de date heap, imediat dupa construirea sa de către procedura Construieste-Heap. (b)-(j) Heap-ul, imediat dupa câte un apel al procedurii Reconstituie-Heap (linia 5 în algoritm). Figura reprezintă valoarea curenta a variabilei *i*. Din heap fac parte doar nodurile din cercurile nehaşurate. (k) Vectorul *A* ordonat, obţinut ca rezultat.

7. Sortarea prin interclasare

Mulţi algoritmi utili au o structura *recursiva*: pentru a rezolva o problema data, aceştia sunt apelaţi de către ei înşişi o data sau de mai multe ori pentru a rezolva subprobleme apropiate.

Aceşti algoritmi folosesc de obicei o abordare de tipul *divide şi stăpâneşte*: ei rup problema de rezolvat în mai multe probleme similare problemei iniţiale, dar de dimensiune mai mica, le rezolva în mod recursiv şi apoi le combina pentru a crea o soluţie a problemei iniţiale.

Pentru metoda de sortare prin interclasare principiul *divide şi stăpâneşte* poate fi privit astfel:

Divide: Împarte şirul de *n* elemente care urmează a fi sortat în doua subşiruri de câte *n*/2 elemente.

Stăpâneşte: Sortează recursiv cele doua subşiruri utilizând sortarea prin interclasare.

Combina: Interclasează cele două subşiruri sortate pentru a produce rezultatul final.

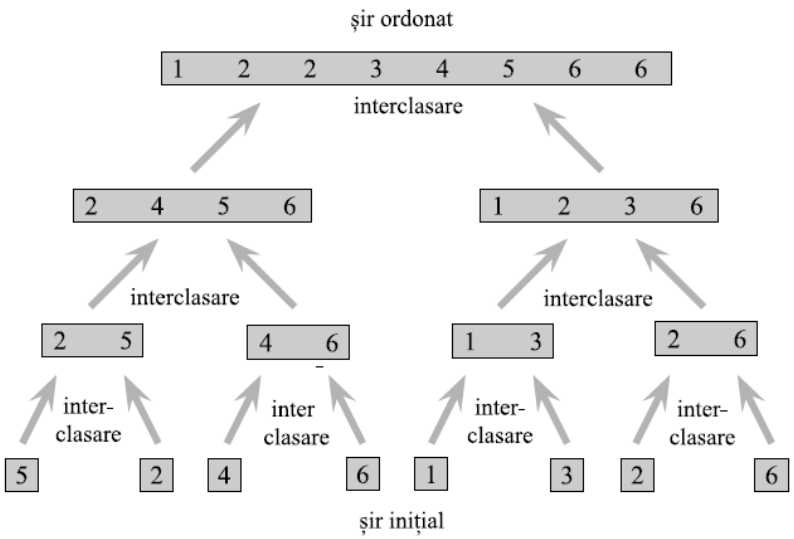

Figura 1.12 Modul de operare al sortării prin interclasare asupra vectorului A = {5, 2, 4, 6, 1, 3, 2, 6}.Lungimile şirurilor sortate, în curs de interclasare, cresc pe măsură ce algoritmul avansează de jos în sus.

Să observăm că recursivitatea se opreşte când şirul de sortat are lungimea 1, caz în care nu mai avem nimic de făcut, deoarece orice sir de lungime 1 este deja sortat.

Operaţia principala a algoritmului de sortare prin interclasare este interclasarea a două şiruri sortate, în pasul denumit mai sus "Combina". Pentru aceasta vom utiliza o procedura auxiliara, Interclaseaza(*A, p, q, r*), unde *A* este un vector şi *p*, *q* şi *r* sunt indici ai vectorului, astfel încât $p \leq q < r$. Procedura presupune că subvectorii $A[p..q]$ şi $A[q + 1..r]$ sunt sortaţi. Ea îi *Interclasează* pentru a forma un subvector sortat care înlocuieşte subvectorul curent $A[p..r]$.

O procedura de tip Interclaseaza are un de execuţie de ordinul $\Theta(n)$, în care $n = r - p + 1$ este numărul elementelor interclasate. Revenind la exemplul nostru cu cărţile de joc, să presupunem că avem doua pachete de cărţi de joc aşezate pe masa cu faţa în sus. Fiecare din cele doua pachete este sortat, cartea cu valoarea cea mai mica fiind deasupra.

Dorim să amestecam cele doua pachete într-un singur pachet sortat, care să rămână aşezat pe masa cu faţa în jos. Pasul principal este acela de a selecta cartea cu valoarea cea mai mica dintre cele doua aflate deasupra pachetelor (fapt care va face ca o noua carte să fie deasupra pachetului respectiv) şi de a o pune cu faţa în jos pe locul în care se va forma pachetul sortat final. Repetam acest procedeu până când unul din pachete este epuizat. În aceasta faza, este suficient să luam pachetul rămas şi să-l punem peste pachetul deja sortat întorcând toate cărţile cu faţa în jos.

Din punctul de vedere al timpului de execuţie, fiecare pas de baza durează un timp constant, deoarece comparam de fiecare data doar doua cărţi. Deoarece avem de făcut cel mult *n* astfel de operaţii elementare, timpul de execuţie pentru procedura Interclaseaza este $\Theta(n)$.

```
Subalgoritm Sorteaza-Prin-Interclasare(A, p, r)
1: daca p < r atunci
2:      q ← [(p + r)/2]
3:      Sorteaza-Prin-Interclasare(A, p, q)
4:      Sorteaza-Prin-Interclasare(A, q + 1, r)
5:      Interclaseaza(A, p, q, r)
```

```
Subalgoritm Interclaseaza(A, p, q, r)
i ← p;
j ← q + 1;
k←1;
cat timp i ≤ q și  j ≤ r executa
        daca  A[i] < A[j] atunci
                C[k] ← A[i]
```

```
                k←k+1
                i←i+1
        altfel
                C[k] ← A[j]
                k←k+1
                j←j+1
cat timp i ≤ q executa
                C[k] ← A[i]
                k←k+1
                i←i+1
cat timp j ≤ r executa
                C[k] ← A[j]
                k←k+1
                j←j+1
k←1;
pentru i ← p, r executa
        A[i] ← C[k]
        k←k+1
```

Utilizam procedura Interclaseaza ca subrutina pentru algoritmul de sortare prin interclasare. Procedura Sorteaza-Prin-Interclasare(*A, p, r*) sortează elementele dinsubvectorul *A*[*p..r*]. Daca *p* ¸ *r*, subvectorul are cel mult un element şi este, prin urmare, deja sortat. Altfel, pasul de divizare este prezent aici prin simplul calcul al unui indice *q* care împarte *A*[*p..r*] în doi subvectori, *A*[*p..q*] conţinând [*n*/2]+1 elemente şi *A*[*q* + 1..*r*] conţinând [*n*/2] elemente.4 Pentru a sorta întregul sir *A* = {*A*[1],*A*[2], ...*A*[*n*]}, vom apela procedura Sorteaza-Prin-Interclasare sub forma Sorteaza-Prin-Interclasare(*A*, 1, *lungime*[*A*]) unde, din nou, *lungime*[*A*] = *n*. Daca analizam modul de operare al procedurii, de jos în sus, când *n* este o putere a lui 2, algoritmul consta din interclasarea perechilor de şiruri de lungime 1, pentru a forma şiruri sortate de lungime 2, interclasarea acestora în şiruri sortate de lungime 4, şi aşa mai departe, până când doua şiruri sortate de lungime *n*/2 sunt interclasate pentru a forma şirul sortat final de dimensiune *n*. Figura 1.12 ilustrează acest proces.

Deşi algoritmul Sorteaza-Prin-Interclasare funcţionează corect când numărul elementelor nu este par, analiza bazata pe recurenţa se simplifica daca presupunem că dimensiunea problemei originale este o putere a lui 2. Fiecare pas de împărţire generează deci doua subşiruri având dimensiunea exact *n*=2.

Pentru a determina recurenţa pentru *T*(*n*), timpul de execuţie al sortării prin interclasare a *n* numere în cazul cel mai defavorabil, vom raţiona în felul următor.

Sortarea prin interclasare a unui singur element are nevoie de un timp constant. Când avem $n > 1$ elemente, vom descompune timpul de execuţie dupa cum urmează:

Divide: La acest pas, se calculează doar mijlocul subvectorului, calcul care are nevoie de un timp constant de execuţie. Astfel, $D(n) = \Theta(1)$.

Stăpâneşte: Rezolvam recursiv doua subprobleme, fiecare de dimensiune $n/2$, care contribuie cu $2T(n/2)$ la timpul de execuţie.

Combina: Am observat deja că procedura Interclaseaza pentru un subvector cu n elemente consuma $\Theta(n)$ timp de execuţie, deci $C(n) = \Theta(n)$.

Când adunam funcţiile $D(n)$ şi $C(n)$ pentru analiza sortării prin interclasare, adunam o funcţie cu timpul de execuţie $\Theta(n)$ cu o funcţie cu timpul de execuţie $\Theta(1)$. Aceasta suma este funcţie liniara în raport cu n, adică are timpul de execuţie $\Theta(n)$. Adăugând aceasta la termenul $2T(n/2)$ de la pasul "Stăpâneşte", obţinem timpul de execuţie $T(n)$ în cazul cel mai defavorabil pentru sortarea prin interclasare:

$$T(n) = \begin{cases} \Theta(1) & \text{dacă } n = 1, \\ 2T(n/2) + \Theta(n) & \text{dacă } n > 1. \end{cases}$$

Pentru numere suficient de mari, sortarea prin interclasare, având timpul de execuţie $\Theta(n \log_2 n)$, este mai performanta decât sortarea prin inserţie, al cărei timp de execuţie în cazul cel mai defavorabil este $\Theta(n^2)$.

8. Sortarea folosind arbore binar de căutare (sau arbore binar ordonat)

Aşa cum sugerează numele său, un arbore binar de căutare este organizat sub forma de arbore binar, aşa cum se observa în figura 1.12. Un astfel de arbore se poate reprezenta printr-o structura de date înlănţuită, în care fiecare nod este un obiect. Pe lângă un câmp *cheie* şi date adiţionale, fiecare obiect nod conţine câmpurile *stânga*, *dreapta* şi *p* care punctează spre (refera) nodurile corespunzătoare fiului stâng, fiului drept şi respectiv părintelui nodului. Daca un fiu sau un părinte lipseşte, câmpul corespunzător acestuia va conţine valoarea nil. Nodul rădăcină este singurul nod din arbore care are valoarea nil pentru câmpul părinte *p*.

Într-un arbore binar de căutare, cheile sunt întotdeauna astfel memorate încât ele satisfac *proprietatea arborelui binar de căutare*:

Fie *x* un nod dintr-un arbore binar de căutare. Daca *y* este un nod din subarborele stâng al lui *x*, atunci *cheie*[*y*] ≤ *cheie*[*x*]. Daca *y* este un nod din subarborele drept al lui *x*, atunci *cheie*[*x*] ≤ *cheie*[*y*].

Astfel, în figura 1.13(a) cheia rădăcinii este 5, iar cheile 2, 3 şi 5 din subarborele stâng nu sunt mai mari decât 5, pe când cheile 7 şi 8 din subarborele drept nu sunt mai mici decât 5. Aceeaşi proprietate se verifica pentru fiecare nod din arbore. De exemplu, cheia 3 din figura 1.13(a) nu este mai mica decât cheia 2 din subarborele său stâng şi nu este mai mare decât cheia 5 din subarborele său drept.

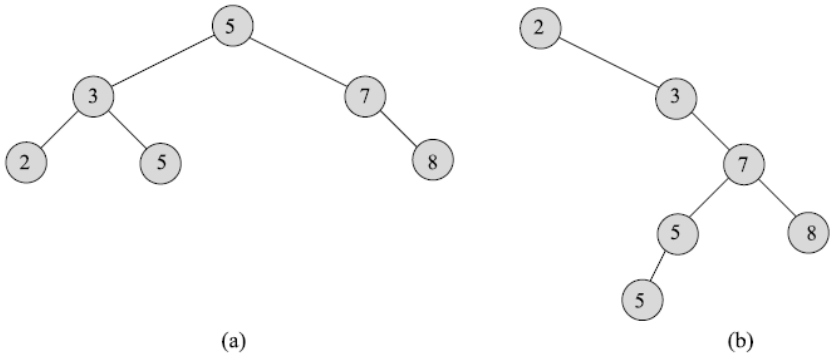

Figura 1.13 Arbori binari de căutare. Pentru orice nod *x*, cheile din subarborele stâng al lui *x* au valoarea mai mica sau egala cu *cheie*[*x*], iar cheile din subarborele drept al lui *x* au valoarea mai mare sau egala cu *cheie*[*x*]. Aceeaşi mulţime de valori se poate reprezenta prin arbori binari de căutare, diferiţi. Timpul de execuţie, în cazul cel mai defavorabil, pentru majoritatea operaţiilor arborilor de căutare este proporţional cu înălţimea arborelui. (a) Un arbore binar de căutare cu 6 noduri şi de înălţime 2. (b) Un arbore binar de căutare mai puţin eficient care conţine aceleaşi chei şi are înălţimea 4.

Proprietatea arborelui binar de căutare ne permite să tipărim toate cheile în ordine crescătoare cu ajutorul unui algoritm recursiv simplu, numit *traversarea arborelui în inordine*.

Numele acestui algoritm deriva din faptul că cheia rădăcinii unui subarbore se tipăreşte între valorile din subarborele său stâng şi cele din subarborele său drept. (Similar, o *traversare a arborelui în preordine* va tipări cheia rădăcinii înaintea cheilor din subarbori, iar o *traversare a arborelui în postordine* va tipări cheia rădăcinii dupa cheile din subarbori). Pentru a folosi procedura următoare în scopul tipăririi tuturor elementelor din arborele binar de căutare *T* o vom apela cu Arbore-Traversare-Inordine(*radacina*[*T*]).

```
Subalgoritm Arbore-Traversare-Inordine(x)
1: daca x ≠ nil atunci
2:     Arbore-Traversare-Inordine(stânga[x])
3:     afiseaza cheie[x]
4:     Arbore-Traversare-Inordine(dreapta[x])
```

Spre exemplu, traversarea în inordine a arborelui afişează cheile fiecăruia dintre arborii binari de căutare din figura 1.13 în ordinea 2, 3, 5, 5, 7, 8. Corectitudinea algoritmului se demonstrează prin inducţie folosind direct proprietatea arborelui binar de căutare. Deoarece dupa apelul iniţial procedura se apelează recursiv de exact doua ori pentru fiecare nod din arbore – o data pentru fiul său stâng şi încă o data pentru fiul său drept – rezulta că este nevoie de un timp $\Theta(n)$ pentru a traversa un arbore binar de căutare cu n noduri.

Crearea arborelui de căutare se realizează prin inserarea de chei noi. Vom folosi procedura Arbore-Insereaza pentru a insera o noua valoare v într-un arbore binar de căutare T. Procedurii i se transmite un nod z pentru care $cheie[z] = v$, $stânga[z]$ = nil şi $dreapta[z]$ = nil. Ea va modifica arborele T şi unele dintre câmpurile lui z astfel încât z va fi inserat pe poziţia corespunzătoare în arbore.

```
Subalgoritm Arbore-Insereaza(T, z)
1: y ← nil
2: x ← radacina[T]
3: cât timp x ≠ nil executa
4:     y ← x
5:     daca cheie[z] < cheie[x] atunci
6:         x ← stânga[x]
7:     altfel
8:         x ← dreapta[x]
9: p[z] ← y
10: daca y = nil atunci
11:     radacina[T] ← z
12: altfel daca cheie[z] < cheie[y] atunci
13:         stânga[y] ← z
14:     altfel
15:         dreapta[y] ← z
```

Algoritmul poate fi uşor implementat şi recursiv.

În cazul cel mai favorabil arborele construi astfel va avea toate frunzele pe 2 nivele consecutive, iar în cel mai defavorabil fiecare nod va fi pe un nivel, ca în figura de mai jos.

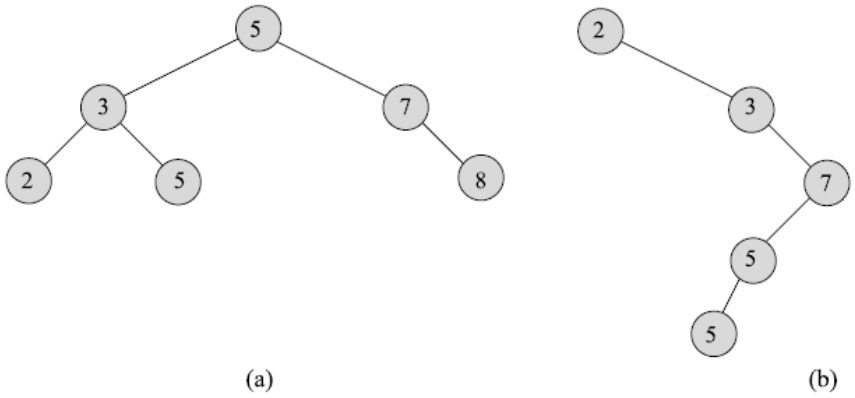

Figura 1.14 Cazul cel mai favorabil (a) şi cel mai defavorabil (b)

Crearea şi utilizarea unui arbore binar de căutare se poate face şi doar cu cele două legături stânga şi dreapta, fără legătura spre nodul părinte.

9. Sortări în timp liniar

Până acum au fost prezentaţi câţiva algoritmi de complexitate $\Theta(n \log_2 n)$ şi $\Theta(n^2)$. $\Theta(n \log_2 n)$ este marginea superioara atinsă de algoritmii de sortare prin interclasare şi heapsort în cazul cel mai defavorabil, respectiv pentru quicksort corespunde în medie.

Aceşti algoritmi au o proprietate interesantă: *ordinea dorita este determinata în exclusivitate pe baza comparaţiilor între elementele de intrare*. Astfel de algoritmi de sortare se numesc *sortări prin comparaţii*. Toţi algoritmii de sortare prezentaţi până acum sunt de acest tip.

Oricare sortare prin comparaţii trebuie să facă, în cel mai defavorabil caz, $\Theta(n \log_2 n)$ comparaţii pentru a sorta o secvenţa de n elemente. Astfel, algoritmul de sortare prin interclasare şi heapsort sunt asimptotic optimale şi nu exista nici o sortare prin comparaţii care să fie mai rapidă decât cu un factor constant.

În secţiunile următoare voi prezenta trei algoritmi de sortare – sortare prin numărarea aparţiilor, ordonare pe baza cifrelor şi ordonare pe grupe – care se executa în timp liniar. Se subînţelege că aceşti algoritmi folosesc, pentru a determina ordinea de sortare, alte operaţii decât comparaţiile.

9.1. Sortarea prin numărarea aparițiilor

Algoritmii prezentaţi anterior sunt relativ mari consumatori de timp. De exemplu, pentru a ordona un şir de 1000 de elemente, numărul comparaţiilor pe care le va executa oricare dintre algoritmii prezentaţi va fi aproximativ de 1 milion.

Un algoritm liniar execută un număr de operaţii proporţional cu numărul elementelor, adică pentru a ordona 1000 de elemente vor fi necesare $c < 1000$ de operaţii, unde c este o constantă. În anumite condiţii asupra datelor de intrare se pot construi algoritmi liniari pentru sortare.

Dacă avem un şir de elemente de tip ordinal care sunt dintr-un interval de cardinalitate nu foarte mare, vom putea realiza o *ordonare liniară*. Corespunzător fiecărei valori întâlnite în şir în timpul prelucrării mărim cu 1 valoarea elementului având indicele (în acest şir de contoare) egal cu valoarea elementului în şirul de ordonat. În final, vom suprascrie în şirul dat atâtea elemente cu valori ai indicilor elementelor diferite de 0 cât este valoarea elementului în acest şir a numerelor de apariţii.

Important este să reţinem particularităţile pe care trebuie să le aibă şirul dat pentru ca această metodă să se poată aplica:

- valorile elementelor trebuie să fie de tip ordinal,
- numărul elementelor mulţimii din care şirul primeşte valori trebuie să fie relativ mic, astfel încât, în funcţie de limbajul de programare în care vom implementa algoritmul să nu se depăşească memoria maximă alocată unui vector.
- valorile posibile în şirul de ordonat trebuie să fie din intervalul $[x..y]$, unde $y - x + 1$ va fi dimensiunea şirului de contoare.

Exemplu

Presupunem că toate valorile din vectorul de sortat sunt numere naturale <=1000. Vom folosi un vector de frecvenţe f.

```
Subalgoritm Ordonare_cu_Şir_de_Frecvenţe(n,a,x,y):
1: x ← 0
2: y ← 1000
3: pentru i=x,y execută
       f[i] ← 0   {iniţializează frecvenţele cu 0}
4: pentru i=1,n execută:
```

```
    f[A[i]] ← f[A[i]] + 1  {măreşte frecvenţa fiecărui element din A}
5: k ← 0
6: pentru i=x,y execută:
7:       pentru j=1,f[i] execută:
8:            k ← k + 1
9:            A[k] ← i     {pune înapoi în A fiecare element de câte ori este frecvenţa lui}
```

Fie şirul (2, 1, 1, 0, 2, 3, 0, 2).

Şirul de frecvenţe construit de algoritmul de mai sus va fi va fi (2, 2, 3, 1).

Corespunzător ultimei secvenţe din algoritm se vor scrie două valori 0 consecutive în *a*, apoi 2 elemente egal cu 1, urmează trei elemente de 2, urmat de o valoare 3: 0, 0, 1, 1, 2, 2, 2, 3.

Deşi este un algoritm liniar, eficienţa lui trebuie bine studiată înainte de a-l aplica. Eficienta depinde atât de numărul de valori din vectorul iniţial, cât şi de valorile propriuzise. Algoritmul depinde liniar de n, numărul de valori din vector, dar capacitatea de memorie necesară pentru vectorul de frecvenţe şi apoi numărul de paşi ai instrucţiunii 6: depinde de limitele în care se află valorile din şir, iar în funcţie de acestea, algoritmul poate deveni mai puţin eficient decât unul cu ordin ce complexitate $\Theta(n^2)$. Un exemplu în acest caz, dacă vrem să ordonăm 100 de numere din intervalul [1, 15000]. O ordonare $\Theta(n^2)$ ar face maxim 10000 de paşi, iar ordonarea cu frecvenţe face 15000, ca să nu mai vorbim de risipa de memorie care se face în cazul folosirii vectorului de frecvenţe.

9.2. Ordonare pe baza cifrelor

Ordonarea pe baza cifrelor este algoritmul folosit de către maşinile de sortare a cartelelor, care se mai găsesc la ora actuală numai în muzeele de calculatoare. Cartelele sunt organizate în 80 de coloane, şi fiecare coloana poate fi perforata în 12 poziţii. Sortatorul poate fi "programat" mecanic să examineze o coloana data a fiecărei cartele dintr-un pachet şi să distribuie fiecare cartela într-una dintre cele 12 cutii, în funcţie de poziţia perforata. Un operator poate apoi să strângă cartelele cutie cu cutie, astfel încât cartelele care au prima poziţie perforata să fie deasupra cartelelor care au a doua poziţie perforata s.a.m.d.

Pentru cifre zecimale sunt folosite numai 10 poziţii în fiecare coloana. (Celelalte două poziţii sunt folosite pentru codificarea caracterelor nenumerice). Un număr având *d* cifre ar ocupa un câmp format din *d* coloane. Întrucât sortatorul de cartele poate analiza numai o singura coloana la un moment dat, problema sortării a *n* cartele în funcţie de un număr având *d* cifre necesita un algoritm de sortare.

Intuitiv, am putea dori să sortam numere în funcţie de *cea mai semnificativa* cifra, să sortam recursiv fiecare dintre cutiile ce se obţin, şi apoi să combinam pachetele în ordine. Din nefericire, întrucât cartelele din 9 dintre cele 10 cutii trebuie să fie păstrate pentru a putea sorta fiecare dintre cutii, aceasta procedura generează teancuri intermediare de cartele care trebuie urmărite.

Ordonarea pe baza cifrelor rezolva problema sortării cartelelor într-un mod care contrazice intuiţia, sortând întâi în funcţie de *cea mai puţin semnificativa* cifra. Cartelele sunt apoi combinate într-un singur pachet, cele din cutia 0 precedând cartelele din cutia 1, iar acestea din urma precedând pe cele din cutia 2 şi aşa mai departe. Apoi întregul pachet este sortat din nou în funcţie de a doua cifra cea mai puţin semnificativa şi recombinat apoi într-o maniera asemănătoare. Procesul continua până când cartelele au fost sortate pe baza tuturor celor *d* cifre.

De remarcat că în acel moment cartelele sunt complet sortate în funcţie de numărul având *d* cifre. Astfel, pentru sortare sunt necesare numai *d* treceri prin lista de numere. În figura 1.16 este ilustrat modul de operare al algoritmului de ordonare pe baza cifrelor pe un “pachet” de şapte numere de câte trei cifre.

Este esenţial ca sortarea cifrelor în acest algoritm să fie stabila. Sortarea realizata de către un sortator de cartele este stabila, dar operatorul trebuie să fie atent să nu schimbe ordinea cartelelor pe măsura ce acestea ies dintr-o cutie, chiar daca toate cartelele dintr-o cutie au aceeaşi cifra în coloana aleasa.

Într-un calculator care funcţionează pe baza de acces secvenţial aleator, ordonarea pe baza cifrelor este uneori utilizata pentru a sorta înregistrările de informaţii care sunt indexate cu chei având câmpuri multiple. De exemplu, am putea dori să sortam date în funcţie de trei parametri: an, luna, zi. Am putea executa un algoritm de sortare cu o funcţie de comparare care, considerând doua date calendaristice, compara anii, şi daca exista o legătură, compara lunile, iar daca apare

din nou o legătură, compara zilele. Alternativ, am putea sorta informaţia de trei ori cu o sortare stabila: prima dupa zi, următoarea dupa luna, şi ultima dupa an.

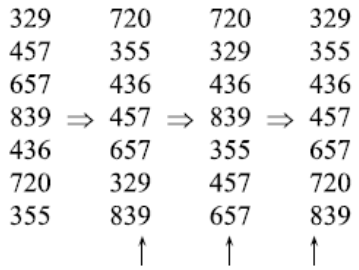

Figura 1.16 Modul de funcţionare al algoritmului de ordonare pe baza cifrelor pe o lista de şapte numere a câte 3 cifre. Prima coloana este intrarea. Celelalte coloane prezintă lista dupa sortări succesive în funcţie de poziţiile cifrelor în ordinea crescătoare a semnificaţiei. Săgeţile verticale indica poziţia cifrei dupa care s-a sortat pentru a produce fiecare lista din cea precedenta.

Pseudocodul pentru algoritmul de ordonare pe baza cifrelor este evident. Următoarea procedura presupune că într-un tablou *A* având *n* elemente, fiecare element are *d* cifre, cifra 1 este cifra cu ordinul cel mai mic, iar cifra *d* este cifra cu ordinul cel mai mare.

```
Subalgoritm Ordonare-Pe-Baza-Cifrelor(A, d)
1: pentru i ← 1, d executa
2:     foloseşte o sortare stabila pentru a sorta tabloul A dupa cifra i
```

Corectitudinea algoritmului de ordonare pe baza cifrelor poate fi demonstrata prin inducţie dupa coloanele care sunt sortate. Analiza timpului de execuţie depinde de sortarea stabila folosita ca algoritm intermediar de sortare. Când fiecare cifra este în intervalul [1, *k*], iar *k* nu este prea mare, sortarea prin numărare este opţiunea evidenta. Fiecare trecere printr-o mulţime de *n* numere a câte *d* cifre se face în timpul $\Theta(n + k)$. Se fac *d* treceri, astfel încât timpul total necesar pentru algoritmul de ordonare pe baza cifrelor este $\Theta(dn + kd)$. Când *d* este constant şi $k = \Theta(n)$, algoritmul de ordonare pe baza cifrelor se executa în timp liniar.

Unii informaticieni considera că numărul biţilor într-un cuvânt calculator este $\Theta(\log_2 n)$. Pentru exemplificare, să presupunem că $d \log_2 n$ este numărul de biţi, unde *d* este o constantă pozitivă.

Atunci, daca fiecare număr care va fi sortat încape într-un cuvânt al calculatorului, îl vom putea trata ca pe un număr având *d* cifre reprezentat în baza *n*. De exemplu, să consideram sortarea a 1 milion de numere având 64 de biţi. Tratând

aceste numere ca numere de patru cifre în baza 216, putem să le sortam pe baza cifrelor doar prin patru treceri, comparativ cu o sortare clasica prin comparaţii de timp $\Theta(n \log_2 n)$ care necesita aproximativ $\log_2 n = 20$ de operaţii pentru fiecare număr sortat. Din păcate, versiunea algoritmului de ordonare pe baza cifrelor care foloseşte sortarea prin numărare ca sortare intermediara stabila nu sortează pe loc, lucru care se întâmpla în cazul multora din sortările prin comparaţii de timp $\Theta(n \log_2 n)$. Astfel, daca se doreşte ca necesarul de memorie să fie mic, atunci este preferabil algoritmul de sortare rapidă.

9.3. Ordonarea pe grupe

Ordonarea pe grupe se executa, în medie, în timp liniar. Ca şi sortarea prin numărare, ordonarea pe grupe este rapidă pentru ca face anumite presupuneri despre datele de intrare. În timp ce sortarea prin numărare presupune că intrarea consta din întregi dintr-un domeniu mic, ordonarea pe grupe presupune că intrarea este generata de un proces aleator care distribuie elementele în mod uniform în intervalul [0, 1).

Principiul algoritmului de ordonare pe grupe este de a împarţi intervalul [0, 1) în n subintervale egale, numite *grupe* (engl. *buckets*) şi apoi să distribuie cele n numere de intrare în aceste grupe. Întrucât datele de intrare sunt uniform distribuite în intervalul [0, 1), nu ne aşteptăm să fie prea multe numere în fiecare grupa. Pentru a obţine rezultatul dorit, sortam numerele din fiecare grupa, apoi trecem prin fiecare grupa în ordine, listând elementele din fiecare.

Pseudocodul pentru ordonarea pe grupe presupune că datele de intrare formează un tablou A având n elemente şi ca fiecare element $A[i]$ satisface relaţia $0 \leq A[i] < 1$. Codul necesita un tablou auxiliar $B[0..n\text{-}1]$ de liste înlănţuite (reprezentând grupele) şi presupune că exista un mecanism pentru menţinerea acestor liste.

```
Subalgoritm Ordonare-Pe-Grupe(A)
1: n ← lungime[A]
2: pentru i ←1, n executa
3:       insereaza A[i] în lista B [[nA[i]]]
4: pentru i ←0, n - 1 executa
5:       sorteaza lista B[i] folosind sortarea prin inserţie
```

6: concateneaza în ordine listele $B[0], B[1], \ldots, B[n-1]$

Pentru a demonstra că acest algoritm funcţionează corect, se considera doua elemente $A[i]$ şi $A[j]$. Daca aceste elemente sunt distribuie în aceeaşi grupa, ele apar în ordinea relativa adecvata în secvenţa de ieşire, deoarece grupa lor este sortata de sortarea prin inserţie. Să presupunem, totuşi, că ele sunt distribuite în grupe diferite. Fie aceste grupe $B[i_0]$ şi respectiv $B[j_0]$, şi să presupunem, fără a pierde din generalitate, că $i_0 < j_0$. Când listele lui B sunt concatenate în linia 6, elementele grupei $B[i_0]$ apar înaintea elementelor lui B $[j_0]$, şi astfel $A[i]$ precede $A[j]$ în secvenţa de ieşire. Deci trebuie să arătăm că $A[i] \leq A[j]$. Presupunând contrariul, avem $i_0 = [nA[i]] \geq [nA[j]] = j_0$, ceea ce este o contradicţie, întrucât $i_0 < j_0$. Aşadar, ordonarea pe grupe funcţionează corect.

Pentru a analiza timpul de execuţie, să observam mai întâi că toate liniile, cu excepţia liniei 5, necesita, în cel mai defavorabil caz, timpul $O(n)$. Timpul total pentru a examina în linia 5 toate grupele este $O(n)$ şi, astfel, singura parte interesanta a analizei este timpul consumat de sortările prin inserţie din linia 5.

Pentru a analiza costul sortărilor prin inserţie, fie *ni* o variabila aleatoare desemnând numărul de elemente din grupa $B[i]$. Întrucât sortarea prin inserţie se executa în timp pătratic, timpul necesar sortării elementelor în grupele B $[i]$ este $E[O(n_i^2)] = O(E[n_i^2])$

În consecinţa, timpul mediu total necesar sortării tuturor elementelor în toate grupele va fi

$$\sum_{i=0}^{n-1} O\left(E\left[n_i^2\right]\right) = O\left(\sum_{i=0}^{n-1} E\left[n_i^2\right]\right)$$

Figura 1.17 ilustrează modul de funcţionare al algoritmului de ordonare pe grupe pe un tablou de intrare cu 10 numere.

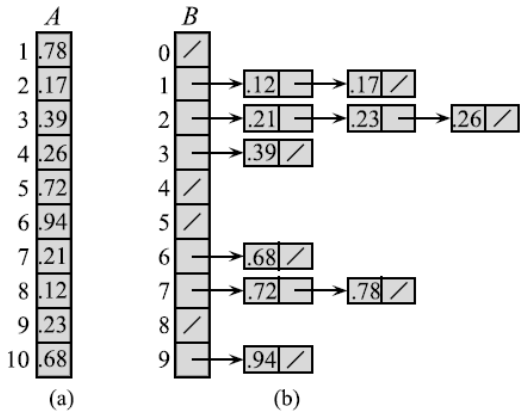

Figura 1.17 Funcţionarea algoritmului Ordonare-Pe-Grupe. (a) Tabloul de intrare *A*[1..10]. (b) Tabloul *B*[0..9] al listelor (reprezentând grupele) sortate dupa linia a cincea a algoritmului. Grupa *I* cuprinde valorile din intervalul [*i*=10, (*i* + 1)=10). Ieşirea sortata consta dintr-o concatenare în ordine a listelor *B* [0], *B* [1] , ..:,*B* [9].

Bibliografie:

1. T.H. Cormen, C.E. Leiserson, R. Rivest : Introducere în algoritmi, Editura Agora, 2001.

2. G. Barbu, I. Văduva, M. Boloşteanu : Bazele Informaticii, Ed. Tehnică, Bucureşti,1997.

3. O. Catrina, Iuliana Cojocaru : Turbo C++, Ed. Teora, 1993.

4. L. Livovschi, H. Georgescu : Bazele Informaticii, Repr. Univ. Bucureşti, 1985.

5. D.E. Knuth : Tratat de programarea calculatoarelor. Algoritmi fundamentali. Sortare şi căutare, Ed. Tehnica, Bucureşti, 1985.

6. D.E. Knuth : Arta programării calculatoarelor – Sortare şi căutare – Ed. Teora, 2002

7. C. Ionescu – Metodica predării informaticii, Univ. "Babeş- Bolyai", Cluj-Napoca, 1998 (curs litografiat)

8. I. Magdaş – Didactica informaticii de la teorie la practică, Editura Clusium 2007

9. C. Masalagiu, I. Asiminoaei, I. Maxim: *Metodica predării informaticii, cursuri de informatică - ghid pentru profesori*.

10. C.Petre, D. Popa, S. Crăciunoiu, C. Iliescu - Metodica predării informaticii şi tehnologiei informaţiei

11. Manualele de informatică pentru liceu