Capitolul 6

Tehnica divizării

Este o tehnică similară tehnicii reducerii însă rezolvarea unei probleme de dimensiune n nu se reduce la rezolvarea unei singure subprobleme de dimensiune mai mică ci la rezolvarea câtorva subprobleme de dimensiuni cât mai apropiate.

6.1 Principiul tehnicii divizării

Tehnica divizării (denumită şi "divide et impera" sau "divide and conquer") se bazează pe descompunerea problemei de rezolvat în două sau mai multe subprobleme *independente*, rezolvarea acestora şi combinarea rezultatelor obţinute. Etapele care se parcurg în aplicarea metodei sunt:

- *Descompunerea în subprobleme.* O problemă de dimensiune n este descompusă în două sau mai multe subprobleme de dimensiune mai mică. Cazul clasic este acela când subproblemele au aceeaşi natură ca şi problema iniţială. Ideal este ca dimensiunile subproblemelor să fie cât mai apropiate (dacă problema de dimensiune n se descompune în k subprobleme e de preferat ca acestea să aibă dimensiuni apropiate de n/k). Pentru ca această tehnică să fie efectivă (să conducă de exemplu la reducerea costului calculului) trebuie ca subproblemele care se rezolvă să fie independente (o aceeaşi subproblemă să nu fie rezolvată de mai multe ori).

- *Rezolvarea subproblemelor.* Fiecare dintre subproblemele independente obţinute prin divizare se rezolvă. Dacă ele sunt similare problemei iniţiale atunci se aplică din nou tehnica divizării. Procesul de divizare continuă până când se ajunge la subprobleme de dimensiune suficient de mică (*dimensiunea critică*, n_c) pentru a fi rezolvate direct.

- *Combinarea rezultatelor.* Pentru a obţine răspunsul la problema iniţială uneori trebuie combinate rezultatele obţinute prin rezolvarea subproblemelor.

Structura generală a unui algoritm elaborat folosind tehnica divizării este descrisă în Algoritmul 6.1. În practică cel mai adesea se utilizează $k = 2$ şi $n_1 = \lfloor n/2 \rfloor$, $n_2 = n - \lfloor n/2 \rfloor$.

Algoritmul 6.1 Structura generală a unui algoritm bazat pe tehnica divizării

```
divizare(P(n))
if n ≤ n_c then
   ⟨ rezolvare directă ⟩
else
   ⟨ descompune P(n) în k subprobleme P(n_1), ... P(n_k) ⟩
   for i ← 1, k do
      divizare(P(n_i))
   end for
   ⟨ compunerea rezultatelor ⟩
end if
```

Pentru a ilustra tehnica considerăm problema determinării maximului dintr-o secvenţă finită de valori reale stocate în tabloul $x[1..n]$. Aplicând ideea divizării rezultă că este suficient să determinăm maximul din subtabloul $x[1..\lfloor n/2 \rfloor]$ şi maximul din subtabloul $x[\lfloor n/2 \rfloor + 1..n]$. Rezultatul va fi cea mai mare dintre valorile obţinute. Varianta recursivă a algoritmului este descrisă în Algoritmul 6.2.

Algoritmul 6.2 Determinarea maximului unui tablou folosind tehnica divizării

```
maxim (real x[s..d])
if s = d then
   max ← x[s]
else
   m ← ⌊(s + d)/2⌋
   max1 ← maxim(x[s..m])
   max2 ← maxim(x[m + 1..d])
   if max1 < max2 then
      max ← max2
   else
      max ← max1
   end if
end if
return max
```

In acest caz se aplică tehnica divizării pentru $k = 2$ şi dimensiunea critică $n_c = 1$. Subproblemele sunt independente, iar compunerea rezultatelor constă în prelucrarea "**if** $max1 < max2$ **then** $max \leftarrow max2$ **else** $max \leftarrow max1$".

Notând cu $T(n)$ numărul de comparaţii efectuate se obţine relaţia de recurenţă:

$$T(n) = \begin{cases} 0 & \text{dacă } n = 1 \\ T(\lfloor n/2 \rfloor) + T(n - \lfloor n/2 \rfloor) + 1 & \text{dacă } n \geq 2 \end{cases}$$

În cazul în care n nu este o putere a lui 2 metoda iteraţiei este mai dificil de aplicat. Pornim de la cazul particular $n = 2^m$ pentru care relaţia de recurenţă conduce la $T(n) = 2T(n/2) + 1$ pentru $n \geq 2$ obţinându-se succesiunea:

$$\begin{array}{lcl|l} T(n) & = & 2T(n/2) + 1 & \\ T(n/2) & = & 2T(n/4) + 1 & \cdot 2^1 \\ T(n/4) & = & 2T(n/8) + 1 & \cdot 2^2 \\ \vdots & & \vdots & \\ T(2) & = & 2T(1) + 1 & \cdot 2^{m-1} \\ T(1) & = & 0 & \cdot 2^m \end{array}$$

Înmulţind relaţiile cu factorii din ultima coloană, însumând relaţiile şi efectuând reducerile se obţine: $T(n) = 1 + 2 + 2^2 + \ldots 2^{m-1} = 2^m - 1 = n - 1$. Acest rezultat este valabil doar pentru $n = 2^m$. Pentru a arăta că este adevărat pentru orice n aplicăm inducţia matematică. Evident $T(1) = 1 - 1 = 0$. Presupunem că $T(k) = k - 1$ pentru orice $k < n$. Rezultă că $T(n) = \lfloor n/2 \rfloor - 1 + n - \lfloor n/2 \rfloor - 1 + 1 = n - 1$, adică algoritmul are complexitate liniară, la fel ca cel obţinut aplicând metoda clasică. Ultimul rezultat poate fi obţinut direct folosind Propoziţia 5.1.

6.2 Analiza complexităţii algoritmilor bazaţi pe tehnica divizării

Analiza complexităţii algoritmilor elaboraţi prin tehnica divizării sau a reducerii se bazează pe un rezultat teoretic important cunoscut sub numele de *Teorema master*.

Presupunem că o problemă de dimensiune n este descompusă în m subprobleme de dimensiuni n/m dintre care este necesar să fie rezolvate $k \leq m$. Considerăm că divizarea şi compunerea rezultatelor au împreună costul $T_{DC}(n)$ iar costul rezolvării în cazul particular este T_0. Cu aceste ipoteze costul corespunzător algoritmului verifică relaţia de recurenţă:

$$T(n) = \begin{cases} T_0 & \text{dacă } n \leq n_c \\ kT(n/m) + T_{DC}(n) & \text{dacă } n > n_c \end{cases}$$

Determinarea lui $T(n)$ pornind de la această recurenţă este uşurată de teorema următoare.

Teorema 6.1 *(Teorema master) Dacă $T_{DC}(n) \in \Theta(n^d)$, $d \geq 0$ atunci:*

$$T(n) \in \begin{cases} \Theta(n^d) & \text{dacă } k < m^d \\ \Theta(n^d lg\ n) & \text{dacă } k = m^d \\ \Theta(n^{\log_m k}) & \text{dacă } k > m^d \end{cases}$$

Teorema rămâne valabilă şi în cazul notaţiei $\mathcal{O}$. Pentru demonstraţie pot fi consultate [3], [11], [12].

Pentru algoritmul `maxim` avem $d = 0$, $m = 2$, $k = 2$, deci se aplică cazul al treilea al teoremei obţinându-se $T(n) = n$.

Teorema poate fi aplicată şi în cazul algoritmilor bazaţi pe tehnica reducerii. Astfel pentru algoritmul căutării binare avem: $T_{DC}(n) = \Theta(1)$ deci $d = 0$, $m = 2$ şi $k = 1$. Se aplică cazul al doilea al teoremei master ($1 = 2^0$) şi se obţine $T(n) = \Theta(\lg n)$.

6.3 Algoritmi de sortare bazaţi pe tehnica divizării

Considerăm din nou problema ordonării crescătoare a unui şir de valori; (x_1,x_2, ..., x_n). Algoritmii bazaţi pe compararea elementelor prezentaţi în Cap. 4 au ordinul de complexitate $\mathcal{O}(n^2)$. Ideea de start o reprezintă încercarea de a reduce această complexitate folosind principiul divizării. Aceasta presupune: (i) descompunerea şirului iniţial în două subsecvenţe; (ii) ordonarea fiecăreia dintre acestea folosind aceeaşi tehnică; (iii) combinarea subsecvenţelor ordonate pentru a obţine varianta ordonată a şirului total.

Dimensiunea critică, sub care problema poate fi rezolvată direct este $n_c = 1$. În acest caz subsecvenţa se reduce la un singur element, implicit ordonat. Este posibil să se folosească şi $1 < n_c \leq 10$ aplicând pentru sortarea acestor subsecvenţe una dintre metodele elementare de sortare (de exemplu metoda inserţiei).

În continuare sunt prezentate două metode de sortare bazate pe principiul divizării: sortarea prin *interclasare* ("mergesort") şi sortarea *rapidă* ("quicksort"). Acestea diferă atât în etapa descompunerii şirului în subşiruri cât şi în recombinarea rezultatelor.

6.3.1 Sortare prin interclasare

Idee.

Şirul $(x_1, x_2, \ldots, x_n)$ se descompune în două subsecvenţe de lungimi cât mai apropiate: $(x_1, \ldots, x_{\lfloor n/2 \rfloor})$ şi $(x_{\lfloor n/2 \rfloor + 1}, \ldots, x_n)$; se ordonează fiecare dintre subsecvenţe aplicând aceeaşi tehnică; se construieşte şirul final ordonat parcurgând cele două subsecvenţe şi preluând elemente din ele astfel încât să fie

respectată relaţia de ordine (această prelucrare se numeşte *interclasare*). Modul de lucru al sortării prin interclasare este ilustrat în figura 6.1.

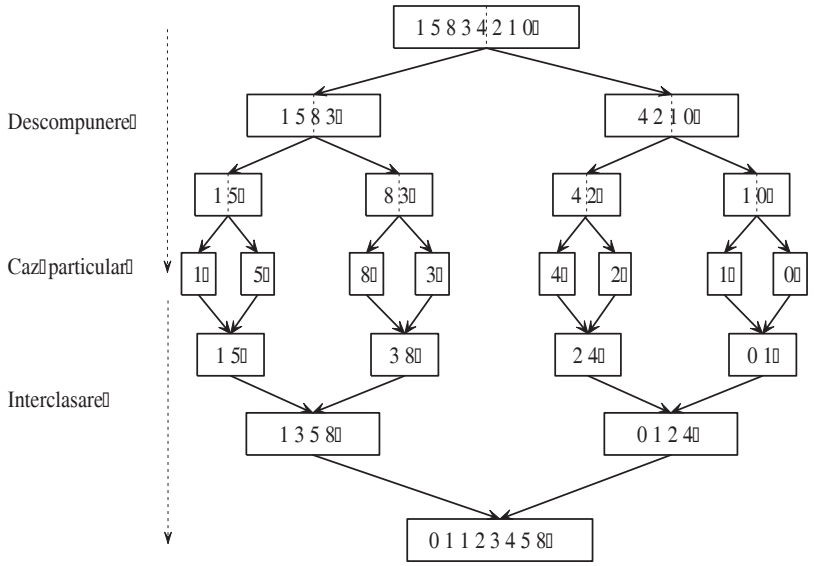

Figura 6.1: Sortare prin interclasare

Structura generală a algoritmului.

Etapele principale ale algoritmului de sortare prin interclasare sunt descrise în Algoritmul 6.3 iar pentru sortarea unui tablou $x[1..n]$ algoritmul trebuie apelat pentru $li = 1$ şi $ls = n$ ($\texttt{mergesort}(x[1..n])$).

Algoritmul 6.3 Structura generală a algoritmului de sortare prin interclasare

```
mergesort(x[li..ls])
if li < ls then
   m = ⌊(li + ls)/2⌋
   x[li..m] ← mergesort(x[li..m])
   x[m + 1..ls] ← mergesort(x[m + 1..ls])
   x[li..ls] ← merge(x[li..m], x[m + 1..ls])
end if
return x[li..ls]
```

Interclasare

Etapa cea mai importantă a prelucrării este reprezentată de interclasare. Scopul acestei prelucrări este construirea unui şir ordonat pornind de la două şiruri ordonate. Ideea prelucrării constă în a parcurge în paralel, folosind două contoare, cele două şiruri şi a compara elementele curente. In şirul final se transferă elementul mai mic dintre cele două iar contorul utilizat pentru parcurgerea şirului din care s-a transferat un element este incrementat. Procesul continuă până când unul dintre şiruri a fost transferat în întregime. Elementele celuilalt şir sunt transferate direct în şirul final.

Există diverse modalităţi de a descrie algoritmic această prelucrare. Una dintre acestea este prezentată în Algoritmul 6.4 în care c reprezintă o zonă suplimentară de memorie.

Algoritmul 6.4 Interclasarea a două şiruri ordonate crescător

```
merge(x[li..m],x[m+1..ls])
i ← li; j ← m+1; k ← 0 // iniţializarea contoarelor
// parcurgerea până la sfârşitul unuia dintre tablouri
while i ≤ m and j ≤ ls do
  if x[i] ≤ x[j] then
    // transfer din tabloul a
    k ← k+1; c[k] ← x[i]; i ← i+1
  else
    // transfer din tabloul b
    k ← k+1; c[k] ← x[j]; j ← j+1
  end if
end while
while i ≤ m do
  // transferul eventualelor elemente rămase în a
  k ← k+1; c[k] ← x[i]; i ← i+1
end while
while j ≤ ls do
  // transferul eventualelor elemente rămase în b
  k ← k+1; c[k] ← x[j]; j ← j+1;
end while
return c[1..k]
```

Precondiţiile Algoritmului 6.4: $\{x[li..m]$ crescător şi $x[m+1..ls]$ crescător$\}$, iar postcondiţia este: $\{c[1..p+q]$ este crescător$\}$ (notăm cu p numărul de elemente din primul tablou şi cu q numărul de elemente din al doilea). Pentru a demonstra că algoritmul asigură satisfacerea postcondiţiei este suficient să se arate că $\{c[1..k]$ este crescător, $k = i+j-2\}$ este proprietate invariantă pentru fiecare dintre cele trei prelucrări repetitive.

Contorizând numărul de comparaţii ($T_C(p,q)$) şi cel de transferuri ale ele-

mentelor ($T_M(p,q)$) se obţine: $T_C(p,q) \in \Omega(\min(p,q))$ (în cazul cel mai favorabil), $T_C(p,q) \in O(p+q-1)$ (în cazul cel mai defavorabil) iar $T_M(p,q) \in \Theta(p+q)$.

Să analizăm cazul interclasării a două tablouri $a[1..p]$ şi $b[1..q]$ (independent de algoritmul de sortare prin interclasare). O variantă de algoritm este cea care foloseşte câte o valoare "santinelă" pentru fiecare dintre tablourile $a[1..p]$ şi $b[1..q]$. Santinelele constau în valori mai mari decât toate valorile prezente în a şi b şi sunt plasate pe poziţiile $p+1$ respectiv $q+1$. În acest caz algoritmul poate fi descris într-o formă mai condensată (Algoritm 6.5) şi atât numărul de comparaţii cât şi numărul de transferuri este $T_C(p,q) = T_M(p,q) = p+q$. Prin urmare în varianta cu valori santinelă se efectuează ceva mai multe comparaţii decât în cea fără astfel de valori însă ordinul de complexitate rămâne liniar în raport cu dimensiunile tablourilor.

Algoritmul 6.5 Varianta cu valori santinelă a algoritmului de interclasare

```
merge2(a[1..p+1],b[1..q+1])
a[p+1] ← ∞; b[q+1] ← ∞; // fixarea santinelelor
i ← 1; j ← 1; //iniţializarea indicilor de parcurgere
for k ← 1, p+q do
   // parcurgerea poziţiilor în tabloul final
   if a[i] ≤ b[j] then
      // preluare din a
      c[k] ← a[i]; i ← i+1
   else
      // preluare din b
      c[k] ← b[j]; j ← j+1
   end if
end for
return c[1..p+q]
```

Analiza complexităţii.

Notând cu $T(n)$ numărul de prelucrări (comparaţii şi transferuri) efectuate de către sortarea prin interclasare şi cu $T_{inter}(n)$ numărul celor efectuate pe parcursul interclasării se obţine relaţia de recurenţă:

$$T(n) = \begin{cases} 0 & n = 1 \\ T(\lfloor n/2 \rfloor) + T(n - \lfloor n/2 \rfloor) + T_{inter}(n) & n > 1 \end{cases}$$

Cum numărul de subprobleme este 2 ($m = 2$) şi ambele trebuie rezolvate ($k = 2$) iar $T_{inter}(n) \in \Theta(n)$ rezultă că se poate aplica cazul doi al Teoremei master ($d = 1$ astfel că $k = m^d$) obţinându-se că sortarea prin interclasare are complexitate $\Theta(n \lg n)$.

Observaţie. Costul redus al prelucrărilor din sortarea prin interclasare este însă contrabalansat de faptul că se utilizează (în etapa de interclasare) o zonă

de manevră de dimensiune proporţională cu cea a tabloului iniţial. In [12] (pag. 256) este descrisă o variantă nerecursivă a sortării prin interclasare caracterizată printr-un număr redus de transferuri de elemente.

6.3.2 Sortare rapidă

Idee

La sortarea prin interclasare descompunerea şirului iniţial în două subsecvenţe se realizează pe baza poziţiei elementelor. Asta face ca elemente cu valori mici (sau mari) să se poată afla în fiecare dintre subşiruri. Din această cauză este necesară combinarea rezultatelor prin interclasare. O altă modalitate de descompunere ar fi aceea în care se ţine cont şi de valoarea elementelor. Scopul urmărit este de a simplifica combinarea rezultatelor. Ideal ar fi ca prin concatenarea şirurilor ordonate să se obţină şirul ordonat în întregime.

Exemplul 6.1 Considerăm şirul $x = (3, 1, 2, 4, 7, 5, 8)$. Se observă că elementul 4 se află pe o poziţie privilegiată ($q = 4$) întrucât toate elementele care îl preced sunt mai mici decât el şi toate cele care îl succed sunt mai mari. Aceasta înseamnă că se află deja pe poziţia finală. Un element $x[q]$ având proprietăţile: $x[i] \leq x[q]$ pentru $i = \overline{1, q-1}$ şi $x[i] \geq x[q]$ pentru $i = \overline{q+1, n}$ se numeşte *pivot*. Existenţa unui pivot permite reducerea sortării şirului iniţial la sortarea subsecvenţelor $x[1..q-1]$ şi $x[q+1..n]$. Nu întotdeauna există un astfel de pivot după cum se observă din subşirul $(3, 1, 2)$. În aceste situaţii trebuie "creat" unul prin modificarea poziţiilor unor elemente. Alteori pivotul, dacă există, se află pe o poziţie care nu permite descompunerea problemei iniţiale în subprobleme de dimensiuni apropiate (de exemplu în subşirul $(7, 5, 8)$ valoarea de pe ultima poziţie poate fi considerată valoare pivot).

Exemplul 6.2 Să considerăm acum şirul $(3, 1, 2, 7, 5, 4, 8)$. Se observă că poziţia $q = 3$ are următoarea proprietate: $x[i] \leq x[j]$ pentru orice $i \in \{1, \ldots, q\}$ şi orice $j \in \{q+1, \ldots, n\}$. În acest caz sortarea şirului iniţial se reduce la sortarea subşirurilor $x[1..q]$ şi $x[q+1..n]$. Poziţia q este numită *poziţie de partiţionare*.

Structura generală

Pornind de la exemplele de mai sus se pot dezvolta două variante de sortare bazate pe descompunerea şirului iniţial în două subsecvenţe: una care foloseşte un element pivot iar alta care foloseşte o poziţie de partiţionare. Acest tip de sortare este cunoscută sub numele de sortare rapidă ("quicksort") şi a fost dezvoltată de Hoare. Structura generală a acestor două variante este descrisă în Algoritmii 6.6 şi 6.7.

Diferenţa dintre cele două variante este mică: în prima variantă în prelucrarea de partiţionare se asigură şi plasarea pe poziţia q a valorii finale pe când în a doua doar se determină poziţia de partiţionare.

Algoritmul 6.6 Sortare rapidă bazată pe element pivot

```
quicksort1 (x[li..ls])
if li < ls then
   q ← partitie1(x[li..ls]
   x[li..q − 1] ← quicksort1(x[li..q − 1])
   x[q + 1..ls] ← quicksort1(x[q + 1..ls])
end if
return x[li..ls]
```

Algoritmul 6.7 Sortare rapidă bazată pe poziţie de partiţionare

```
quicksort2 (x[li..ls])
if li < ls then
   q ← partitie2(x[li..ls]
   x[li..q] ← quicksort1(x[li..q])
   x[q + 1..ls] ← quicksort1(x[q + 1..ls])
end if
return x[li..ls]
```

Partiţionare

Este prelucrarea cea mai importantă a algoritmului. Discutăm separat cele două variante deşi după cum se va vedea diferenţele dintre ele sunt puţine.

Considerăm problema identificării în $x[1..n]$ a unui pivot, $x[q]$ cu proprietatea că $x[i] \leq x[q]$ pentru $i < q$ şi $x[i] \geq x[q]$ pentru $i > q$. După cum s-a văzut în exemplul anterior nu întotdeauna există un pivot. În aceste situaţii se creează unul. Ideea este următoarea: se alege o valoare arbitrară dintre cele prezente în şir şi se rearanjează elementele şirului (prin interschimbări) astfel încât elementele mai mici decât această valoare să fie în prima parte a şirului iar cele mai mari în a doua parte a şirului. În felul acesta se determină şi poziţia q pe care trebuie plasată valoarea. Prelucrarea este descrisă în Algoritmul 6.8.

Exemplul 6.4 Considerăm şirul $(1, 7, 5, 3, 8, 2, 4)$. Iniţial $i = 0$, $j = 7$ iar $v = 4$. Succesiunea prelucrărilor este:
Etapa 1. După execuţia primului ciclu **repeat** se obţine $i = 2$ (căci $7 > 4$) iar după execuţia celui de al doilea se obţine $j = 6$. Cum $i < j$ se interschimbă $x[2]$ cu $x[6]$ obţinându-se $(1, 2, 5, 3, 8, 7, 4)$.
Etapa 2. Parcurgerea în continuare de la stânga la dreapta conduce la $i = 3$ iar de la dreapta la stânga la $j = 4$. Cum $i < j$ se interschimbă $x[3]$ cu $x[4]$ şi se obţine $(1, 2, 3, 5, 8, 7, 4)$.
Etapa 3. Continuând parcurgerile în cele două direcţii se ajunge la $i = 4$ şi $j = 3$.
Cum $i > j$ se iese din prelucrarea repetitivă exterioară, iar elementele $x[4]$ şi $x[7]$ se interschimbă obţinându-se $(1, 2, 3, 4, 8, 7, 5)$, poziţia pivotului fiind 4.

Algoritmul 6.8 Determinarea poziţiei pivotului

```
partitie1(x[li..ls])
v ← x[ls] // se alege valoarea pivotului
i ← li − 1 // contor pentru parcurgere de la stânga la dreapta
j ← ls // contor pentru parcurgere de la dreapta la stânga
while i < j do
  repeat
    i ← i + 1
  until x[i] ≥ v
  repeat
    j ← j − 1
  until x[j] ≤ v
  if i < j then
    x[i] ↔ x[j]
  end if
end while
x[i] ↔ x[ls]
return i
```

Această poziţie asigură o partiţionare echilibrată a şirului. Această situaţie nu este însă obţinută întotdeauna.

Exemplul 6.5 Să considerăm şirul $(4, 7, 5, 3, 8, 2, 1)$. Aplicând acelaşi algoritm, după prima etapă se obţine: $i = 1$, $j = 1$ (a se observa că în acest caz fără a folosi o poziţie suplimentară cu rol de fanion, $x[0] = v$ condiţia de oprire a celui de-al doilea **repeat** nu este niciodată satisfăcută), şirul devine $(1, 7, 5, 3, 8, 2, 4)$ iar poziţia pivotului este $q = 1$. In acest caz s-a obţinut o partiţionare dezechilibrată (şirul se descompune într-un subşir vid, pivotul aflat pe prima poziţie şi un subşir constituit din celelalte elemente). Se poate remarca că pentru primul ciclu **repeat** $x[ls]$ joacă rolul unui fanion.

Observaţii.

(i) Inegalităţile de tip $\geq$ şi $\leq$ din ciclurile **repeat** fac ca, în cazul unor valori egale, să se obţină o partiţionare echilibrată şi nu una dezechilibrată cum s-ar întâmpla dacă s-ar folosi inegalităţi stricte. În acelaşi timp dacă s-ar utiliza inegalităţi stricte valoarea v nu ar putea fi folosită ca fanion.

(ii) Problema nesatisfacerii condiţiei de oprire pentru ciclul după j poate să apară doar în cazul în care $li = 1$ şi atunci când v este cea mai mică valoare din şir. Este de preferat să se folosească valoare fanion decât să se modifice condiţia de oprire de la ciclu (de exemplu $x[j] \leq v$ **and** $j < i$).

(iii) La ieşirea din prelucrarea repetitivă exterioară (**while**) indicii i şi j satisfac una dintre relaţiile: $i = j$ sau $i = j + 1$. Ultima interschimbare

asigură plasarea valorii fanion pe poziţia sa finală.

Pentru a justifica corectitudinea algoritmului `partiţie1` considerăm aserţiunea: {dacă $i < j$ atunci $x[k] \leq v$ pentru $k = \overline{li, i}$ iar $x[k] \geq v$ pentru $k = \overline{j, ls}$ iar dacă $i \geq j$ atunci $x[k] \leq v$ pentru $k = \overline{li, i}$ iar $x[k] \geq v$ pentru $k = \overline{j+1, ls}$}. Cum precondiţia este $li < ls$, deci $i < j$ şi postcondiţiile sunt $\{x[k] \leq x[i]$ pentru $k = \overline{1, i-1}$, $x[k] \geq x[i]$ pentru $k = \overline{i+1, ls}\}$, se poate arăta că aserţiunea de mai sus este invariantă în raport cu prelucrarea repetitivă **while** iar după efectuarea ultimei interschimbări implică postcondiţiile.

Considerăm acum cealaltă variantă de partiţionare. Aceasta diferă de prima în special prin faptul că permite ca valoarea pivotului să participe la interschimbări. În plus el nu este plasat la sfârşit pe poziţia finală fiind necesară astfel sortarea subtablourilor $x[li..q]$ şi $x[q+1..ls]$. Aceasta face să nu mai fie necesară plasarea pe poziţia 0 a unui fanion întrucât se creează în mod natural fanioane pentru fiecare dintre cele două cicluri **repeat**. Această variantă este descrisă în Algoritmul 6.9.

Algoritmul 6.9 Determinarea poziţiei de partiţionare

```
partitie2(x[li..ls])
v ← x[li] // se alege valoarea pivotului
i ← li − 1 // contor pentru parcurgere de la stânga la dreapta
j ← ls + 1 // contor pentru parcurgere de la dreapta la stânga
while i < j do
  repeat
    i ← i + 1
  until x[i] ≥ v
  repeat
    j ← j − 1
  until x[j] ≤ v
  if i < j then
    x[i] ↔ x[j]
  end if
end while
return j
```

În acest caz se poate arăta că aserţiunea {dacă $i < j$ atunci $x[k] \leq v$ pentru $k = \overline{li, i}$ iar $x[k] \geq v$ pentru $k = \overline{j, ls}$ iar dacă $i \geq j$ atunci $x[k] \leq v$ pentru $k = \overline{li, i-1}$ iar $x[k] \geq v$ pentru $k = \overline{j+1, ls}$} este invariantă în raport cu ciclul **while** iar la ieşirea din ciclu (când $i = j$ sau $i = j+1$) implică $x[k] \leq v$ pentru $k = \overline{li, j}$ şi $x[k] \geq v$ pentru $k = \overline{j+1, ls}$ adică postcondiţia $x[k1] \leq x[k2]$ pentru orice $k1 \in \{li, \ldots, j\}$, $k2 \in \{j+1, \ldots, ls\}$.

De remarcat că dacă se alege ca pivot $x[ls]$, varianta `quicksort2` nu va funcţiona corect putând conduce la situaţia în care unul dintre subtablouri este vid (ceea ce provoacă o succesiune nesfârşită de apeluri recursive, întrucât

nu se mai reduce dimensiunea problemei). În cazul în care se doreşte ca $x[ls]$ să fie valoarea pivot este necesar ca apelurile recursive să se facă pentru: `quicksort2`$(x[li..q-1])$ şi `quicksort2`$(x[q..ls])$.

Ca urmare a reorganizării şirului în etapa determinării pivotului sau a poziţiei de partiţionare, poziţia relativă a elementelor având aceeaşi valoare a cheii de sortare nu este neapărat conservată. Prin urmare algoritmul de sortare rapidă nu este stabil.

Analiza complexităţii

Analizăm complexitatea variantei `quicksort1`. Pe parcursul partiţionării, numărul de comparaţii depăşeşte în general numărul de interschimbări, motiv pentru care vom analiza doar numărul de comparaţii efectuate. Pentru un şir de lungime n numărul de comparaţii efectuate în cele două cicluri **repeat** din `partiţionare1` este $n+i-j$ (i şi j fiind valorile finale ale contoarelor). Astfel, dacă $i = j$ se vor efectua n comparaţii iar dacă $i = j + 1$ se vor efectua $n + 1$ comparaţii.

Influenţa majoră asupra numărului de comparaţii efectuate de către algoritmul `quicksort1` o are raportul dintre dimensiunile celor două subşiruri obţinute după partiţionare. Cu cât dimensiunile celor două subşiruri sunt mai apropiate cu atât se reduce numărul comparaţiilor efectuate. Astfel cazurile cele mai favorabile corespund partiţionării echilibrate iar cele mai puţin favorabile partiţionării dezechilibrate.

Analiza în cazul cel mai favorabil. Presupunem că la fiecare partiţionare a unui şir de lungime n se obţin două subsecvenţe de lungimi $\lfloor n/2 \rfloor$ respectiv $n - \lfloor n/2 \rfloor - 1$ iar numărul de comparaţii din partiţionare este n. În aceste condiţii putem considera că marginea superioară a numărului de comparaţii satisface o relaţie de recurenţă de forma $T(n) = 2T(n/2) + n$ ceea ce conduce prin teorema master la o complexitate de ordin $n\lg n$. Prin urmare în cazul cel mai favorabil ordinul de complexitate a algoritmului `quicksort1` este $n\lg n$, adică algoritmul aparţine lui $\Omega(n\lg n)$.

Analiza în cazul cel mai defavorabil. Presupunem că la fiecare partiţionare se efectuează $n + 1$ comparaţii şi că se generează un subşir vid şi unul de lungime $n-1$. Atunci relaţia de recurenţă corespunzătoare este $T(n) = T(n-1)+n+1$. Aplicând metoda iteraţiei se obţine $T(n) = (n + 1)(n + 2)/2 - 3$. Se obţine astfel că în cazul cel mai defavorabil complexitatea este pătratică, adică sortarea rapidă aparţine clasei $\mathcal{O}(n^2)$. Spre deosebire de metodele elementare de sortare pentru care un şir iniţial sortat conducea la un număr minim de prelucrări în acest caz tocmai aceste situaţii conduc la costul maxim (pentru un şir deja ordonat la fiecare partiţionare se obţine pivotul pe ultima poziţie).

Pentru a evita partiţionarea dezechilibrată au fost propuse diverse variante de alegere a valorii pivotului, una dintre acestea constând în selecţia a trei valori

din şir şi alegerea ca valoare a pivotului a medianei acestor valori (valoarea aflată pe a doua poziţie în tripletul ordonat).

Analiza în cazul mediu. Se bazează pe ipoteza că toate cele n poziţii ale şirului au aceeaşi probabilitate de a fi selectate ca poziţie pivot (probabilitatea este în acest caz $1/n$). Presupunând că la fiecare partiţionare a unui şir de lungime n se efectuează $n+1$ comparaţii, numărul de comparaţii efectuate în cazul în care poziţia pivotului este q satisface: $T_q(n) = T(q-1) + T(n-q) + n + 1$. Prin urmare numărul mediu de comparaţii satisface:

$$\begin{aligned} T_m(n) &= \tfrac{1}{n}\textstyle\sum_{q=1}^{n} T_q(n) = (n+1) + \tfrac{1}{n}\textstyle\sum_{q=1}^{n}(T_m(q-1) + T_m(n-q)) \\ &= (n+1) + \tfrac{2}{n}\textstyle\sum_{q=1}^{n} T_m(q-1). \end{aligned}$$

Scăzând între ele relaţiile:

$$nT_m(n) = 2\sum_{q=1}^{n} T_m(q-1) + n(n+1)$$

$$(n-1)T_m(n-1) = 2\sum_{q=1}^{n-1} T_m(q-1) + (n-1)n$$

se obţine relaţia de recurenţă pentru T_m:

$$nT_m(n) = (n+1)T_m(n-1) + 2n$$

Aplicând metoda iteraţiei rezultă:

$$\begin{array}{lcl|l} T_m(n) & = & \dfrac{n+1}{n}T_m(n-1) + 2 & \\ T_m(n-1) & = & \dfrac{n}{n-1}T_m(n-2) + 2 & \cdot\frac{n+1}{n} \\ T_m(n-2) & = & \dfrac{n-1}{n-2}T_m(n-3) + 2 & \cdot\frac{n+1}{n-1} \\ \vdots & & \vdots & \\ T_m(2) & = & \dfrac{3}{2}T_m(1) + 2 & \cdot\frac{n+1}{3} \\ T_m(1) & = & 0 & \end{array}$$

iar prin însumare se obţine:

$$\begin{aligned} T_m(n) &= 2(n+1)(1/n + 1/(n-1) + \ldots + 1/3) + 2 \\ &= 2(n+1)\textstyle\sum_{i=3}^{n}\frac{1}{i} + 2 \simeq 2(n+1)(\ln n - \ln 3) + 2. \end{aligned}$$

Prin urmare numărul mediu de comparaţii este de ordinul $2n\ln n \simeq 1.38n\lg n$. Aceasta înseamnă că în cazul mediu sortarea rapidă este doar cu 38% mai costisitoare decât în cazul cel mai favorabil.