Curs 7: Tehnici de elaborare a algoritmilor: reducere si divizare
- Motivaţie
- Algoritmi recursivi
- Tehnica reducerii
- Tehnica divizării
- Exerciţii

1 Motivaţie

Considerăm problema calculului lui x^n pentru $x > 0$ şi $n = 2^m$, $m \geq 1$ fiind un număr natural. Algoritmul general pentru calculul lui x^n (pentru un n natural arbitrar):

putere1 (x,n)
$p \leftarrow 1$
FOR $i \leftarrow 1, n$ DO
    $\|p \leftarrow p * x$
RETURN p

are complexitatea $\Theta(n)$. Dacă însă se foloseşte ipoteza $n = 2^m$ se observă că $x^n = x^{n/2} \cdot x^{n/2}$ căci $x^{2^m} = x^{2^{(m-1)}} \cdot x^{2^{(m-1)}}$. Prin urmare pentru a calcula x^n este suficient să se calculeze $x^{n/2}$ şi să se ridice la pătrat. La rândul său calculul lui $x^{n/2}$ poate fi redus la calculul lui $x^{n/4}$ şi la o ridicare la pătrat ş.a.m.d. Descompunerea poate continua până se ajunge la x^2 al cărui calcul este simplu. Algoritmul poate fi descris prin:

putere2 (x,m)
$p \leftarrow x$
FOR $i \leftarrow 1, m$ DO
    $\|p \leftarrow p * p$
RETURN p

Folosind ca invariant pentru prelucrarea repetitivă afirmaţia: $\{ p = x^{2^{(i-1)}} \}$ se poate demonstra cu uşurinţă că algoritmul *putere2* calculează x^{2^m}. Pe de altă parte se observă că prelucrarea este de complexitate $\Theta(m) = \Theta(\lg(n))$. Reducerea complexităţii derivă din faptul că la fiecare etapă se calculează valoarea unui singur factor din cei doi, întrucât sunt identici. Ideea de rezolvare a acestei probleme este comună tehnicilor de reducere şi divizare care se bazează la rezolvarea unei probleme de dimensiune n pe rezolvarea uneia sau mai multor probleme similare dar de dimensiune mai mică. Reducerea dimensiunii continuă până când se ajunge la o problemă de dimensiune suficient de mică pentru a putea fi rezolvată direct (de exemplu $n = 2$ sau $m = 1$). O descriere a acestei metode de rezolvare care ilustrează mai bine ideea reducerii dimensiunii este:

putere3 (x,n)
IF $n = 2$ THEN RETURN $x * x$
    ELSE
        $\|p \leftarrow putere3(x, n/2)$
        $\|$RETURN $p * p$

Dacă se transmit ca parametri valorile x şi m algoritmul pentru calculul lui x^{2^m} poate fi descris prin:

putere4 (x,m)
IF $m = 1$ THEN RETURN $x * x$
ELSE
$\|p \leftarrow putere4(x, m-1)$
$\|$RETURN $p * p$

Ultimii doi algoritmi prezintă o particularitate: în cadrul prelucrărilor pe care le efectuează există şi un auto-apel (în cadrul funcţiei se apelează aceeaşi funcţie). Astfel de algoritmi se numesc *recursivi*.

Exemplul de mai sus ilustrează faptul că folosind ideea reducerii rezolvării unei probleme la rezolvarea unei probleme similare dar de dimensiune mai mică *poate* conduce la reducerea complexităţii. Pe de altă parte există probleme pentru care abordarea rezolvării în această manieră este mai uşoară conducând la algoritmi intuitivi.

Trebuie menţionat totodată că nu întotdeauna tehnicile din această categorie conduc la o reducere a complexităţii în cazul în care algorimul este implementat pe o maşină secvenţială. Un exemplu în acest sens îl reprezintă calculul factorialului (după cum se va demonstra mai târziu, aplicând tehnica reducerii se obţine un algoritm de complexitate $\Theta(n)$ la fel ca prin aplicarea tehnicii forţei brute).

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. Algoritmul poate fi descris în manieră recursivă după cum urmează:

maxim $(x[s..d])$
IF $s = d$ THEN $max \leftarrow x[s]$
ELSE
$\|m \leftarrow \lfloor (s+d)/2 \rfloor$
$\|max1 \leftarrow maxim(x[s..m])$
$\|max2 \leftarrow maxim(x[m+1..n)$
$\|$IF $max1 < max2$ THEN $max \leftarrow max2$ ELSE $max \leftarrow max1$
RETURN max

Nu este dificil de observat că ordinul de complexitate al acestui algoritm este tot $\Theta(n)$ ca şi în cazul algoritmului clasic (afirmaţia va fi justificată în secţiunea dedicată analizei algoritmilor recursivi). Acest lucru este adevărat în cazul în care maşina pe care va fi executat algoritmul este una secvenţială. Dacă însă maşina este *paralelă* (la un moment dat pot fi efectuate simultan mai multe prelucrări) atunci printr-o astfel de abordare se poate ajunge la câştig de timp.

2 Algoritmi recursivi

Ultimii algoritmi prezentaţi în secţiunea anterioară fac parte din categoria algoritmilor recursivi. Aceştia sunt utilizaţi pentru a descrie prelucrări ce se pot specifica prin ele însele. Un algoritm recursiv este caracterizat prin:

- *Condiţie de oprire.* Specifică situaţia în care rezultatul se poate obţine prin calcul direct fără a mai fi necesară apelarea aceluiaşi algoritm.
- *Auto-apel.* Se apelează cel puţin o dată pentru alte valori ale parametrilor. Valorile parametrilor corespunzătoare succesiunii de apeluri trebuie să asigure apropierea de satisfacerea condiţiei de oprire.

Ca urmare a cascadei de auto-apeluri un algoritm recursiv realizează de fapt o prelucrare repetitivă, chiar dacă aceasta nu este explicită. Un exemplu simplu de prelucrare repetitivă descrisă recursiv este cea corespunzătoare determinării cmmdc a două numere naturale nenule, $a \geq b > 0$. Algoritmul bazat pe operaţia de împărţire poate fi descris prin:

---

cmmdcr1 (a, b)
IF $b = 0$ THEN $rez \leftarrow a$
ELSE $rez \leftarrow cmmdcr1(b, a \text{ MOD } b)$
RETURN rez

---

iar cel bazat pe operaţia de scădere prin:

---

cmmdcr2 (a, b)
IF $a = b$ THEN $rez \leftarrow a$
ELSE IF $a > b$ THEN $rez \leftarrow cmmdcr2(b, a - b)$
ELSE $rez \leftarrow cmmdcr2(a, b - a)$
RETURN rez

Exemplele de mai sus se caracterizează prin recursivitate *simplă* şi *directă*. Un algoritm recursiv este considerat simplu recursiv dacă conţine un singur auto-apel şi multiplu recursiv dacă conţine două sau mai multe auto-apeluri (de exemplu, algoritmul *maxim* din secţiunea anterioară).

Există şi posibilitatea ca algoritmul să nu se auto-apeleze direct ci indirect prin intermediul altui algoritm. De exemplu algoritmul *A1* apelează algoritmul *A2* iar acesta apelează algoritmul *A1*. În acest caz este vorba de recursivitate indirectă.

Noţiunea de recursivitate poate fi utilizată şi în contextul definirii unor concepte. De exemplu conceptul de expresie aritmetică poate fi definit astfel: "o expresie aritmetică este constituită din operanzi şi operatori; un operand poate fi o constantă, o variabilă sau o *expresie*". Descrierea recursivă permite specificarea unor structuri infinite folosind un set finit de reguli.

Algoritmii recursivi sunt adecvaţi pentru rezolvarea problemelor sau prelucrarea datelor descrise în manieră recursivă.

Arbori de apel. Pentru a ilustra modul de lucru al unui algoritm iterativ poate fi util să se reprezintă grafic structura de apeluri şi reveniri cu returnarea rezultatului obţinut. În cazul unui algoritm recursiv arbitrar structura de apeluri este una ierarhică conducând la un *arbore de apel*. În cazul recursivităţii simple arborele de apel degenerează într-o structură liniară.

Verificarea corectitudinii algoritmilor recursivi. Dacă relaţia de recurenţă care descrie legătura dintre soluţiilor corespunzătoare diferitelor instanţe ale problemei este corectă atunci şi algoritmul care o implementează este corect. Pe de altă parte întrucât un algoritm recursiv specifică o prelucrare repetitivă implicită pentru a demonstra corectitudinea acestuia este suficient să se arate că:

- Există o aserţiune referitoare la starea algoritmului care are proprietăţile: (i) este adevărată pentru cazul particular (când e satisfăcută condiţia de oprire); (ii) este adevărată la revenirea din apelul recursiv şi efectuarea prelucrărilor locale; (iii) implică postcondiţia.
- Condiţia de oprire este satisfăcută după o succesiune finită de apeluri recursive.

Pentru algoritmul *cmmdcr1* o aserţiune invariantă este $\{rez = cmmdc(a, b)\}$. Pentru cazul particular $b = 0$ avem $rez = a = cmmdc(a, 0) = cmmdc(a, b)$. Dacă $rez = cmmdc(a, b)$ înainte de apelul recursiv întrucât $cmmdc(a, b) = cmmdc(b, aMODb)$ rezultă că $rez = cmmdc(a, b)$ după apelul recursiv. Pentru algoritmul *cmmdcr2* proprietatea invariantă este tot $\{rez = cmmdc(a, b)\}$.

Figura 1: Structura de apel pentru algoritmul *cmmdcr1*

În ambele situaţii condiţia de oprire va fi satisfăcută după un număr finit de apeluri recursive, datorită proprietăţilor restului unei împărţiri întregi.

Analiza complexităţii algoritmilor recursivi. Considerăm o problemă de dimensiune n şi algoritmul recursiv de forma:

---

algrec (n)
IF $n = n_0$ THEN P_1
ELSE $algrec(h(n))$

---

cu $h(n)$ o funcţie descrescătoare cu proprietatea că există k cu $h^{(k)}(n) = (h \circ h \circ \ldots \circ h)(n) = n_0$. Dacă prelucrarea P_1 are cost constant (c_0) iar determinarea lui $h(n)$ are costul c atunci costul algoritmului *algrec* poate fi descris prin:

$$T(n) = \begin{cases} c_0 & \text{dacă } n = n_0 \\ T(h(n)) + c & \text{dacă } n > n_0 \end{cases}$$

Prin urmare timpul de execuţie al unui algoritm recursiv satisface o relaţie de recurenţă. Pentru determinarea expresiei lui $T(n)$ pornind de la relaţia de recurenţă se poate folosi una dintre metodele:

- *Metoda substituţiei (directe).* Pornind de la relaţia de recurenţă se intuieşte forma generală a lui $T(n)$ după care se demonstrează prin inducţie matematică validitatea expresiei lui $T(n)$.
- *Metoda iteraţiei (substituţiei inverse).* Se scrie relaţia de recurenţă pentru n, $h(n)$, $h(h(n))$, ..., n_0 după care se substituie succesiv $T(h(n))$, $T(h(h(n)))$ ş.a.m.d. Din relaţia obţinută, pe baza unor calcule algebrice rezultă expresia lui $T(n)$. Metoda mai este cunoscută şi ca metoda substituţiei inverse.

Exemplul 1. Să considerăm cazul $h(n) = n - 1$.

Prin metoda iteraţiei se obţine:

$$\begin{array}{lcl} T(n) & = & T(n-1)+c \\ T(n-1) & = & T(n-2)+c \\ \vdots & & \vdots \\ T(n_0+1) & = & T(n_0)+c \\ T(n_0) & = & c_0 \end{array}$$

Prin însumarea tuturor relaţiilor şi reducerea termenilor corespunzători se obţine: $T(n) = c(n - n_0) + c_0$ pentru $n > n_0$.

Exemplul 2. Considerăm algoritmul recursiv (*putere3*) pentru calculul puterii x^{2^m}. Dacă notăm numărul înmulţirilor efectuate cu $T(n)$ se obţine relaţia de recurenţă:

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

Aplicând tehnica iteraţiei obţinem:

$$\begin{array}{lcl} T(n) & = & T(n/2)+1 \\ T(n/2) & = & T(n/2^2)+1 \\ \vdots & & \vdots \\ T(4) & = & T(2)+1 \\ T(2) & = & 1 \end{array}$$

Cum numărul relaţiilor de mai sus este m, prin însumarea tuturor şi reducerea termenilor corespunzători se obţine $T(n) = m = \lg n$.

Exemplul 3. Analizăm algoritmul *maxim* descris în secţiunea anterioară. 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:

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

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 $T(n) = \lfloor n/2 \rfloor - 1 + n - \lfloor n/2 \rfloor - 1 + 1 = n - 1$.

Observaţie. Se poate demonstra că dacă $T(n) \in \Theta(f(n))$ pentru $n = 2^m$, $T(n)$ este crescătoare pentru argumente mari ($n \geq n_0$) iar $f(n)$ are proprietatea $f(cn) \in \Theta(f(n))$ pentru c o constantă (în acest caz se spune despre f că este netedă) atunci $T(n) \in \Theta(f(n))$ pentru orice n.

Condiţiile de mai sus ($T(n)$ crescătoare pentru valori mari ale lui n şi $f(cn) \in \Theta(f(n))$ sunt satisfăcute în marea majoritate a situaţiilor practice.

3 Tehnica reducerii

Principiu. Tehnica reducerii se bazează pe relaţia care există între soluţia unei probleme şi soluţia unei instanţe de dimensiune redusă a aceleiaşi probleme. De regulă reducerea dimensiunii se bazează pe scăderea unei constante (în majoritatea situaţiilor 1) din dimensiunea problemei.

Calculul factorialului. Cel mai simplu exemplu este cel al calculului factorialului. Relaţia de la care se porneşte este:

$$n! = \begin{cases} 1 & \text{dacă } n = 0 \\ (n-1)! \cdot n & \text{dacă } n > 0 \end{cases}$$

Algoritmul poate fi descris în variantă recursivă astfel:

---

factorial (n)
IF $n = 0$ THEN $f = 1$
ELSE $f = factorial(n-1) * n$
RETURN f

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

$$T(n) = \begin{cases} 0 & \text{dacă } n = 0 \\ T(n-1) + 1 & \text{dacă } n > 0 \end{cases}$$

Din această relaţie se poate intui că $T(n) = n$. Demonstrăm acest lucru prin inducţie matematică după n. Pentru $n = 1$ se obţine $T(1) = T(0) + 1 = 1$. Presupunem că $T(n-1) = n-1$. Din relaţia de recurenţă vom obţine $T(n) = T(n-1) + 1 = n$. Deci într-adevăr $T(n) = n$. Acelaşi rezultat se obţine aplicând tehnica iteraţiei:

$$\begin{array}{lcl} T(n) & = & T(n-1) + 1 \\ T(n-1) & = & T(n-2) + 1 \\ \vdots & & \vdots \\ T(1) & = & T(0) + 1 \\ T(0) & = & 0 \end{array}$$

Prin însumarea tuturor relaţiilor şi efectuarea reducerilor se obţine: $T(n) = n$.

Generarea permutărilor. Să considerăm problema generării permutărilor de ordin n. Există mai multe moduri de a aplica tehnica reducerii. O variantă porneşte de la ideea că o permutare de ordin n se poate obţine dintr-o permutare de ordin $n-1$ prin plasarea valorii n pe toate cele n poziţii posibile. Astfel dacă $n = 3$ există două permutări de ordin $n-1 = 2$: $(1,2)$ şi $(2,1)$. Pentru fiecare dintre acestea valoarea 3 poate fi inserată în fiecare dintre cele trei poziţii posibile: prima, a doua şi a treia conducând la $(3,1,2)$, $(1,3,2)$, $(1,2,3)$ respectiv la $(3,2,1)$, $(2,3,1)$, $(2,1,3)$. Această idee ar corespunde unei abordări ascendente ("bottom-up" - pornind de la o permutare de ordin dat se construieşte o permutare de ordin imediat superior).

Pentru o abordare descendentă ("top-down") a problemei (o permutare de ordin n se specifică prin permutări de ordin $n-1$) să observăm că pentru a genera toate permutările de ordin n este

suficient să plasăm pe poziţia n succesiv toate valorile posibile (din $\{1, 2, \ldots, n\}$) şi pentru fiecare valoare astfel plasată să generăm toate permutările corespunzătoare valorilor aflate pe primele $n-1$ poziţii. Pentru a le genera pe acestea se folosesc permutările de ordin $n-2$ până se ajunge la permutări de ordin 1 (o singură permutare care constă chiar din valoarea aflată pe poziţia 1).

Pentru a descrie algoritmul să presupunem că permutările se vor obţine într-un tablou $x[1..n]$ accesat în comun de către toate (auto)apelurile algoritmului şi iniţializat astfel încât $x[i] = i$. În momentul în care x conţine o permutare ea este afişată.

```
permutari(k)
IF k = 1 THEN WRITE x[1..n]
ELSE
    ||FOR i ← 1, k DO
    ||    ||x[i] ↔ x[k]
    ||    ||permutari(k − 1)
    ||    ||x[i] ↔ x[k]
```

Aplicând acest algoritm, permutările de ordin 3 se obţin în ordinea următoare: $(2, 3, 1)$, $(3, 2, 1)$, $(3, 1, 2)$, $(1, 3, 2)$, $(2, 1, 3)$, $(1, 2, 3)$.

Pentru a analiza complexitatea algoritmului contorizăm numărul de interschimbări efectuate şi observăm că satisface:

$$T(k) = \begin{cases} 0 & \text{dacă } k = 1 \\ k(T(k-1)+2) & \text{dacă } k > 1 \end{cases}$$

Aplicăm tehnica iteraţiei şi obţinem

$$\begin{array}{lcl|l} T(k) & = & kT(k-1)+2k & \\ T(k-1) & = & (k-1)T(k-2)+2(k-1) & \cdot k \\ T(k-2) & = & (k-2)T(k-3)+2(k-2) & \cdot k(k-1) \\ \vdots & & \vdots & \\ T(2) & = & 2T(1)+2 & \cdot k(k-1)\cdots 3 \\ T(1) & = & 0 & \cdot k(k-1)\cdots 3\cdot 2 \end{array}$$

Înmulţind fiecare relaţie cu factorii specificaţi în ultima coloană şi însumând toate relaţiile se obţine: $T(k) = k! + 2(k(k-1)\cdots 4 + \ldots k(k-1) + k)$. Prin urmare pentru $k = n$ se obţine $T(n) = \in \Theta(n!)$.

Turnurile din Hanoi. Se consideră trei vergele plasate vertical şi identificate prin s (sursă), d (destinaţie) şi i (intermediar). Pe vergeaua s se află dispuse n discuri în ordinea descrescătoare a razelor (primul disc are raza maximă). Se cere să se transfere toate discurile pe vergeaua d astfel încât să fie plasate în aceeaşi ordine. Se poate folosi vergeaua i ca intermediar cu restricţia că în orice moment peste un disc se află doar discuri de rază mai mică. Ideea rezolvării este: "se transferă $n-1$ discuri de pe s pe i folosind d ca intermediar; se transferă discul rămas pe s direct pe d; se transferă cele $n-1$ discuri de pe i pe d folosind s ca intermediar". Această idee poate fi descrisă simplu în manieră recursivă:

```
hanoi(n,s,d,i)
IF n = 1 THEN s → d
ELSE
    ||hanoi(n − 1, s, i, d)
    ||s → d
    ||hanoi(n − 1, i, d, s)
```

În algoritmul *hanoi* primul argument specifică numărul de discuri ce vor fi transferate, al doilea indică vergeaua sursă, al treilea vergeaua destinaţie iar ultima pe cea folosită ca intermediar. Prelucrarea $s \to d$ specifică faptul că va fi transferat discul de pe vergeaua s pe vergeaua d. Structura apelurilor recursive pentru $n = 3$ este ilustrată în fig. 2. Succesiunea mutărilor este obţinută parcurgând blocurile haşurate de la stânga la dreapta: $s \to d$, $s \to i$, $d \to i$, $s \to d$, $i \to s$, $i \to d$, $s \to d$.

Figura 2: Structura de apel pentru algoritmul *hanoi* când $n = 3$

Pentru a analiza complexitatea contorizăm numărul de mutări de discuri. Se observă că:

$$T(n) = \begin{cases} 1 & \text{dacă } n = 1 \\ 2T(n-1) + 1 & \text{dacă } n > 1 \end{cases}$$

Prin metoda iteraţiei se obţine:

$$\begin{array}{lcl|l} T(n) & = & 2T(n-1)+1 & \\ T(n-1) & = & 2T(n-2)+1 & \cdot 2^1 \\ T(n-2) & = & 2T(n-3)+1 & \cdot 2^2 \\ \vdots & & \vdots & \\ T(2) & = & 2T(1)+1 & \cdot 2^{n-2} \\ T(1) & = & 1 & \cdot 2^{n-1} \end{array}$$

Însumând relaţiile rezultă $T(n) = 1 + 2^1 + 2^2 + \ldots + 2^{n-1} = 2^n - 1$ adică algoritmul are ordinul de complexitate $\Theta(2^n)$.

Înmulţirea "à la russe". Reducerea dimensiunii se poate realiza nu doar prin scăderea unei constante ci şi prin împărţirea la o constantă. Cazul cel mai frecvent este acela al împărţirii dimensiunii problemei la 2. Un exemplu simplu în acest sens este cel al înmulţirii "à la russe". Regula de calcul în acest caz este:

$$a \cdot b = \begin{cases} 0 & \text{dacă } a = 0 \\ \frac{a}{2} \cdot 2b & \text{dacă a este par} \\ \frac{a-1}{2} \cdot 2b + b & \text{dacă a este impar} \end{cases}$$

Pornind de la această relaţie algoritmul poate fi descris prin:

```
produs(a,b)
IF a = 0 THEN RETURN 0
ELSE
    ||IF a MOD 2 = 0 THEN RETURN produs(a  DIV  2,2*b)
    ||ELSE RETURN produs((a-1) DIV 2,2*b)+b
```

Pentru a analiza complexitatea algoritmului considerăm că dimensiunea problemei este reprezentată de perechea (a, b) iar operaţia dominantă este împărţirea lui a la 2 (celelalte operaţii: înmulţirea lui b cu 2 şi adunarea nu se efectuează de mai multe ori). Astfel pentru timpul de execuţie $T(a, b)$ se poate scrie relaţia de recurenţă:

$$T(a,b) = \begin{cases} 0 & \text{dacă } a = 0 \\ T(a/2, 2b) + 1 & \text{dacă } a > 0 \end{cases}$$

Aplicăm metoda iteraţiei pentru cazul particular $a = 2^k$:

$$\begin{array}{lcl} T(2^k, b) & = & T(2^{k-1}, 2b) + 1 \\ T(2^{k-1}, 2b) & = & T(2^{k-2}, 2^2 b) + 1 \\ \vdots & & \vdots \\ T(2, 2^{k-1}b) & = & T(1, 2^k b) + 1 \\ T(1, 2^k b) & = & T(0, 2^{k+1} b) + 1 \\ T(1, 2^{k+1} b) & = & 0 \end{array}$$

Însumând relaţiile rezultă $T(a, b) = k + 1 = \lg a + 1$. Se observă că timpul de execuţie depinde doar de valoarea lui a şi pentru $a = 2^k$ avem $T(a) \in \Theta(\lg a)$. Întrucât $T(a)$ este crescătoare pentru valori mari ale lui a iar $\lg ca = \lg a + \lg c \in \Theta(\lg a)$ rezultă că rezultatul este valabil şi pentru cazul general.

4 Tehnica divizării

Principiu. Tehnica divizării (denumită şi "divide et impera" sau "divide and conquer") constă în:

- *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 natura 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 pentru subprobleme.

Structura generală a unui algoritm elaborat folosind tehnica divizării este:

divizare($P(n)$)
IF $n \leq n_c$ THEN $\langle$ rezolvare directă $\rangle$
ELSE
  ‖$\langle$ descompune $P(n)$ în k subprobleme $P_1(n_1), \ldots P_k(n_k)$ $\rangle$
  ‖FOR $i \leftarrow 1, k$
  ‖ ‖*divizare*($P_i(n_i)$)
  ‖$\langle$ compunerea rezultatelor $\rangle$

Exemple. În practică cel mai adesea se utilizează $k = 2$ şi $n_1 = \lfloor n/2 \rfloor$, $n_2 = n - \lfloor n/2 \rfloor$. Algoritmul pentru determinarea maximului prezentat în prima secţiune se bazează pe 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 $max \leftarrow max1$".

Să considerăm problema căutării unei valori v într-un tablou $a[1..n]$ ordonat crescător. Tehnica divizării se poate aplica aici astfel:

- se alege un element $a[j]$ din $a[1..n]$ care se compară cu v;
- dacă $v = a[j]$ atunci a fost găsit elementul căutat;
- dacă $v < a[j]$ atunci căutarea continuă în subtabloul $a[1..j-1]$ altfel ea continuă în subtabloul $a[j+1..n]$.

Alegerea cea mai naturală pentru j este să fie cât mai aproape de mijlocul tabloului. Pe de altă parte, se observă că pentru această problemă doar una dintre cele două subprobleme trebuie rezolvată. Dimensiunea critică este în acest caz $n_c = 1$ (tablou constituit dintr-un singur element).

Algoritmul elaborat în acest mod, numit algoritmul *căutării binare*, poate fi descris astfel:

cautbin1($a[s..d]$,v)
IF $s > d$ THEN RETURN F
ELSE
  ‖$m \leftarrow (s + d)$ DIV 2
  ‖IF $a[m] = v$ THEN RETURN A
  ‖ELSE
  ‖ ‖IF $v < a[m]$ THEN RETURN *cautbin*1($a[s..m-1], v$)
  ‖ ‖ELSE RETURN *cautbin*1($a[m+1..d], v$)

cu s şi d delimitând zona din tablou unde se concentrează la un moment dat căutarea. La început se caută în întreg tabloul astfel că la primul apel avem $s = 1$ şi $d = n$. Algoritmul poate fi descris şi în variantă iterativă:

cautbin2($a[1..n]$,v)
$s \leftarrow 1$
$d \leftarrow n$
$gasit \leftarrow False$
WHILE $(s \leq d) \wedge (gasit = False)$
  ‖$m \leftarrow \lfloor (s + d)/2 \rfloor$
  ‖IF $a[m] = v$ THEN $gasit \leftarrow True$
  ‖ELSE
  ‖ ‖IF $v < a[m]$ THEN $d \leftarrow m - 1$
  ‖ ‖ELSE $s \leftarrow m + 1$
RETURN gasit

Pentru a evita compararea de două ori a valorii căutate cu elemente ale tabloului ($a[m] = v$ şi $v < a[m]$) algoritmul poate fi rescris în forma următoare:

cautbin3($a[1..n]$,v)
$s \leftarrow 1$
$d \leftarrow n$
WHILE $s < d$
  ‖$m \leftarrow \lfloor (s+d)/2 \rfloor$
  ‖IF $v \leq a[m]$ THEN $d \leftarrow m$
  ‖ELSE $s \leftarrow m+1$
IF v=a[s] THEN RETURN True
ELSE RETURN FALSE

Pentru a analiza algoritmul căutării binare să considerăm că $T(n)$ reprezintă numărul maxim de comparaţii efectuate asupra elementelor tabloului (se atinge în cazul cel mai defavorabil, când v nu se află în tablou). Relaţia de recurenţă corespunzătoare este:

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

Pentru stabili valoarea lui $T(n)$ să analizăm pentru început cazul particular $n = 2^m$. Aplicaând metoda iteraţiei obţinem:

$$\begin{array}{lcll} T(2^m) & = & T(n) & = & T(n/2)+1 \\ T(2^{m-1}) & = & T(n/2) & = & T(n/4)+1 \\ \vdots & & \vdots & & \\ T(2^1) & = & T(2) & = & T(1)+1 \\ T(2^0) & = & T(1) & = & 1 \end{array}$$

Însumând cele $m+1$ relaţii se obţine $T(n) = m+1 = \lg n + 1$, pentru $n = 2^m$. Pentru n arbitrar relaţia se demonstrează prin inducţie matematică. Presupunem că $T(k) = \lfloor \lg k \rfloor + 1$ pentru orice $k < n$ şi arătăm că $T(n) = \lfloor \lg n \rfloor + 1$. Tratăm separat cazurile: (i) $n = 2k$; (ii) $n = 2k+1$. În primul caz se obţine:

$$T(n) = T(k) + 1 = \lfloor \lg k \rfloor + 2 = \lfloor \lg k + 1 \rfloor + 1 = \lfloor lg(2k) \rfloor + 1 = \lfloor \lg n \rfloor + 1.$$

În al doilea caz obţinem:

$$T(n) = T(k) + 1 = \lfloor \lg k \rfloor + 2 = \lfloor \lg k + 1 \rfloor + 1 = \lfloor \lg(2k) \rfloor + 1 = \lceil \lg(2k+1) \rceil = \lfloor \lg n \rfloor + 1.$$

folosind relaţiile $\lfloor \lg n \rfloor + 1 = \lceil lg(n+1) \rceil$ pentru $n \in N$ şi $\lfloor x \rfloor + 1 = \lceil x \rceil$ pentru $x \notin N$.

Prin urmare căutarea binară este de complexitate $O(\lg n)$.

Observaţie. Algoritmul căutării binare poate fi mai degrabă văzut ca un exemplu de aplicare a tehnicii reducerii (prin împărţirea problemei în două subprobleme dintre care doar una se rezolvă).

Metoda master. Reprezintă o tehnică de analiză a algoritmilor elaboraţi prin tehnica divizării. 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 master. Dacă $T_{DC}(n) = \Theta(n^d)$, $d \geq 0$ atunci:

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

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

5 Exerciţii

1. Propuneţi un algoritm mai eficient decât *putere1* pentru calculul puterii x^n pentru un n, număr natural arbitrar. Stabiliţi ordinul de complexitate al algoritmului propus.

2. Propuneţi un algoritm pentru calculul lui A^n, cu A matrice pătratică de ordin n care să aibă o complexitate mai mică decât $\Theta(n^4)$.

3. Folosind relaţia $C_n^k = C_{n-1}^k + C_{n-1}^{k-1}$ scrieţi un algoritm recursiv pentru calculul lui C_n^k. Analizaţi complexitatea algoritmului propus şi comparaţi ordinul de complexitate cu cel al algoritmului bazat pe relaţia: $C_n^k = n!/(k!(n-k)!)$.

4. Modificaţi algoritmul *maxim* astfel încât să permită determinarea atât a valorii maxime cât şi a celei minime. Stabiliţi numărul de comparaţii efectuate. Algoritmul obţinut este mai eficient decât cel care constă în determinarea separată a minimului şi a maximului ?

5. Scrieţi un algoritm care să implementeze următoarea idee pentru determinarea simultană a valorii minime şi maxime dintr-un tablou: "Se grupează elementele tabloului în $\lceil n/2 \rceil$ perechi. Pentru fiecare pereche se transferă valoarea minimă într-o mulţime cu minime iar valoarea maximă într-o mulţime cu maxime. Se aplică aceeaşi prelucrare asupra celor două mulţimi eliminând din mulţimea minimelor valorile mari şi din cea a maximelor valorile mici. Procesul continuă până când aceste mulţimi conţin câte un element. Analizaţi algoritmul propus.

6. Scrieţi un algoritm de generare a permutărilor bazat pe abordarea "bottom-up" şi analizaţi complexitatea acestuia.

   Indicaţie. Algoritmul va fi apelat cu $perm(1)$ şi poate fi descris prin:

```
perm(k)
IF k = n + 1 THEN WRITE x[1..n]
ELSE
    ||FOR i ← 1, k DO
    ||    ||x[i] ↔ x[k]
    ||    ||perm(k + 1)
    ||    ||x[i] ↔ x[k]
```

7. Modificaţi algoritmul de căutare binară astfel încât să returneze poziţia valorii căutate în cazul în care aceasta e prezentă în tablou şi -1 în caz contrar.

8. Modificaţi algoritmul de sortare prin inserţie astfel încât căutarea poziţiei în care se inserează elementul $x[i]$ în subtabloul $x[1..i-1]$ se bazează pe tehnica căutării binare. Analizaţi complexitatea algoritmului obţinut. Se modifică ordinul de complexitate în raport cu algoritmul clasic de inserţie ?