Capitolul 5

Tehnica reducerii

Tehnica reducerii se bazează pe ideea de a reduce rezolvarea unei probleme de dimensiune dată la rezolvarea unei probleme similare de dimensiune mai mică. De cele mai multe ori dimensiunea este redusă prin scăderea unei constante (de la problema de dimensiune n se ajunge la problema de dimensiune $n-1$, cum se întâmplă la calculul factorialului) însă poate fi redusă şi prin împărţirea la o constantă (ca în exemplul cu înmulţirea "à la russe").

5.1 Motivaţie

Considerăm problema calculului lui x^n pentru $x > 0$ şi $n = 2^m$, $m \geq 1$ fiind un număr natural. Metoda clasică, bazată pe tehnica forţei brute, pentru calculul lui x^n (pentru un n natural arbitrar) este descrisă în algoritmul `putere1` şi se observă uşor că are complexitatea $\Theta(n)$. Dacă însă se analizează cazul particular $n = 2^m$ 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. O astfel de variantă ascendentă de calcul a lui x^{2^m} este descrisă în algoritmul `putere2`.

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 care se bazează la rezolvarea unei probleme de dimensiune n pe rezolvarea unei 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 rezol-

Algoritmul 5.1 Calculul puterii unui număr folosind tehnica forţei (`putere1`) brute respectiv tehnica reducerii (`putere2`)

`putere1` (**real** x,**integer** n)	`putere2` (**real** x,**integer** m)
$p \leftarrow 1$	$p \leftarrow x$
for $i \leftarrow 1, n$ **do**	**for** $i \leftarrow 1, m$ **do**
$\quad p \leftarrow p * x$	$\quad p \leftarrow p * p$
end for	**end for**
return p	**return** p

vată direct (de exemplu $n = 2$ sau $m = 1$). O descriere a acestei metode de rezolvare care ilustrează mai bine ideea reducerii dimensiunii este cea în care se foloseşte o abordare descendentă (algoritmul `putere3`). Dacă se transmit ca parametri valorile x şi m algoritmul pentru calculul lui x^{2^m} poate fi descris ca în `putere4`.

Algoritmul 5.2 Variante recursive ale algoritmilor de calcul a puterii unui număr

`putere3` (**real** x,**integer** n)	`putere4` (**real** x,**integer** m)
if $n = 2$ **then**	**if** $m = 1$ **then**
$\quad$ **return** $x * x$	$\quad$ **return** $x * x$
else	**else**
$\quad p \leftarrow$`putere3`$(x, n/2)$	$\quad p \leftarrow$ `putere4`$(x, m-1)$
$\quad$ **return** $p * p$	$\quad$ **return** $p * p$
end if	**end if**

Algoritmii `putere3` şi `putere4` 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. În plus există probleme pentru care abordarea rezolvării în această manieră este mai uşoară conducând la algoritmi mai intuitivi.

Trebuie menţionat totodată că nu întotdeauna tehnicile din această categorie conduc la o reducere a complexităţii. Un exemplu în acest sens îl reprezintă calculul factorialului (după cum se va ilustra în Exemplul 5.1, aplicând tehnica reducerii se obţine un algoritm de complexitate $\Theta(n)$ la fel ca prin aplicarea tehnicii forţei brute).

5.2 Analiza algoritmilor 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:

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

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 celui mai mare divizor comun a două numere naturale nenule, $a \geq b > 0$. Variantele recursive de determinare a celui mai mare divizor comun se bazează pe proprietăţile pe care le satisface acesta şi anume:

$$cmmdc(a,b) = \begin{cases} a & b = 0 \\ cmmdc(b, a \text{ MOD } b) & b \neq 0 \end{cases}$$

respectiv

$$cmmdc(a,b) = \begin{cases} a & a = b \\ cmmdc(b, a-b) & a > b \\ cmmdc(a, b-a) & a < b \end{cases}$$

Aceste relaţii pot fi uşor descrise în manieră recursivă (vezi Algoritmul 5.3).

Algoritmul 5.3 Variante recursive ale algoritmului lui Euclid

```
cmmdcr1 (integer a, b)
if b = 0 then
  rez ← a
else
  rez ← cmmdcr1(b, a MOD b)
end if
return rez
```

```
cmmdcr2 (integer a, b)
if a = b then
  rez ← a
else if a > b then
  rez ← cmmdcr2(b, a − b)
else
  rez ← cmmdcr2(a, b − a)
end if
return rez
```

Exemplele prezentate până acum se caracterizează prin recursivitate *simplă* şi *directă*. Un algoritm este considerat simplu recursiv dacă se auto-apelează o

singură dată şi multiplu recursiv dacă conţine două sau mai multe auto-apeluri (de exemplu, algoritmul `hanoi` din secţiunea următoare).

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*.

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

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

5.2.1 Arbori de apel

Pentru a ilustra modul de lucru al unui algoritm recursiv poate fi util să se reprezinte 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 (Figura 5.1).

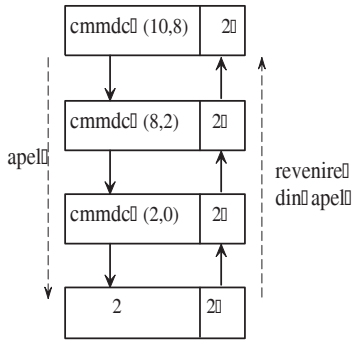

Figura 5.1: Structura de apel pentru algoritmul `cmmdcr1`

5.2.2 Verificarea corectitudinii algoritmilor recursivi

Dacă relaţia de recurenţă ce descrie legătura dintre soluţiile corespunzătoare instanţelor de diferite dimensiuni 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) rămâne adevărată la revenirea din apelul recursiv şi după efectuarea eventualelor prelucrări locale; (iii) pentru valorile de apel ale parametrilor de intrare 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 $cmmdc(a, b) = cmmdc(b, a \text{ MOD } b)$ rezultă că $rez = cmmdc(a, b)$ este adevărată şi după apelul recursiv. Pentru algoritmul `cmmdcr2` proprietatea invariantă este de asemenea $\{rez = cmmdc(a, b)\}$. În ambele situaţii condiţia de oprire va fi satisfăcută după un număr finit de apeluri recursive, datorită proprietăţilor restului unei împărţiri întregi.

5.2.3 Analiza complexităţii algoritmilor recursivi

Considerăm o problemă de dimensiune n şi algoritmul recursiv având forma generală descrisă în `algrec`.

`algrec`(n)
if $n = n_0$ **then**
  P_1
else
  $algrec(h(n))$
end if

În algoritmul `algrec`, $h(n)$ este o funcţie descrescătoare cu proprietatea că există k cu $h^{(k)}(n) = (h \circ h \circ \ldots \circ h)(n) = n_0$ (aceasta înseamnă că după k apeluri recursive este satisfăcută condiţia de oprire). 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 următoarea relaţie de recurenţă:

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

Pentru determinarea expresiei lui $T(n)$ pornind de la relaţia de recurenţă se poate folosi una dintre metodele următoare:

- *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.* 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 5.1 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 5.2 Considerăm algoritmul recursiv (`putere3`) pentru calculul puterii $x^n = x^{2^m}$. 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 $T(n) = m = \lg n$.

5.3 Principiul tehnicii reducerii

Tehnica reducerii se bazează pe relaţia ce 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, însă poate fi realizată şi prin împărţirea dimensiunii problemei la o altă valoare (în acest caz abordarea este similară cu cea de la tehnica divizării). Să considerăm câteva exemple.

Exemplul 5.3 (*Calculul factorialului*) Cel mai simplu exemplu este cel al calculului factorialului. Relaţia de la care se porneşte este:

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

Varianta recursivă de calcul a factorialului este descrisă în algoritmul `factrec`.

Algoritmul 5.4 Variantă recursivă pentru calculul factorialului

```
factrec (integer n)
if n = 0 then
   f ← 1
else
   f ← factrec(n − 1) ∗ n
end if
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: $T(n) = n$.

Exemplul 5.4 (*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:

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

Metoda este descrisă în variantă recursivă în Algoritmul 5.5.

Algoritmul 5.5 Variantă recursivă pentru înmulţirea "à la russe"

```
prodrec(integer a,b)
if a = 0 then
  return 0
else if a MOD 2 = 0 then
  return prodrec(a DIV 2, 2 * b)
else
  return prodrec((a − 1) DIV 2, 2 * b) + b
end if
```

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, adică înmulţirea lui b cu 2 şi adunarea se efectuează de acelaşi număr de 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 $a = 2^k$:

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

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 $T(a) \in \Theta(\lg a)$. Extinderea acestui rezultat pentru valori arbitrare ale lui n se bazează pe următorul rezultat cunoscut sub numele de regula funcţiilor netede ("smoothness rule").

Propoziţia 5.1 *Dacă $T(n)$ satisface următoarele proprietăţi:*

(i) $T(n)$ este crescătoare pentru valori ale lui n suficient de mari;

(ii) $T(n) \in \Theta(f(n))$ pentru $n = 2^m$;

iar $f(n)$ satisface condiţia de netezime ($f(cn) \in \Theta(f(n))$ pentru orice constantă pozitivă, c) atunci $T(n) \in \Theta(f(n))$ pentru orice valoare a lui n.

Rezultatul de mai sus este adevărat şi pentru celelalte două clase de complexitate ($\mathcal{O}$ şi Ω). Condiţiile din Propoziţia 5.1 ($T(n)$ crescătoare pentru valori mari ale lui n şi $f(cn) \in \Theta(f(n))$) sunt satisfăcute în toate cazurile cand f este polinomială. Condiţia $f(cn) \in \Theta(f(n))$ nu este însă satisfăcută pentru funcţii f cu creştere rapidă, de exemplu $f(n) = a^n$ cu $a > 1$.

Revenind la exemplul 5.4, 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 obţinut pentru cazul $a = 2^k$ este valabil şi pentru cazul general.

5.4 Aplicaţii

5.4.1 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 succesivă a 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).

In descrierea algoritmului 5.6 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 $x[i] = i$ pentru fiecare poziţie i. În momentul în care x conţine o permutare, aceasta este afişată. Algoritmul va fi descris pentru un parametru de intrare generic, k, şi va fi apelat pentru $k = n$.

Algoritmul 5.6 Generarea tuturor permutărilor de ordin n

```
permutari(integer 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]
  end for
end if
```

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 acesta 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 $T(n) \in \Omega(n!)$ şi $T(n) \in \mathcal{O}(n \cdot n!)$. Ordinul mare de complexitate este de aşteptat având în vedere că se generează $n!$ permutări.

5.4.2 Problema turnurilor din Hanoi

O problemă clasică ce permite ilustrarea ideii de la tehnica reducerii este problema turnurilor 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 (Algoritmul 5.7).

Algoritmul 5.7 Problema turnurilor din Hanoi

```
hanoi(integer n,s,d,i)
if n = 1 then
   s → d (transferă de la s la d)
else
   hanoi(n − 1, s, i, d)
   s → d (transferă de la s la d)
   hanoi(n − 1, i, d, s)
end if
```

În algoritmul `hanoi` primul argument specifică numărul de discuri ce vor fi transferate, al doilea indică vergeaua sursă, al treilea vergeaua destinaţie iar ultimul pe cea folosită ca intermediar. Prelucrarea $s \to d$ specifică faptul că va fi transferat discul de pe vergeaua s pe vergeaua d. Pentru cazul a trei discuri procesul de transfer este ilustrat în fig 5.2 iar structura apelurilor recursive în fig. 5.3. Succesiunea mutărilor este obţinută parcurgând blocurile haşurate de la stânga la dreapta: $s \to d$, $s \to i$, $d \to i$, $s \to d$, $i \to s$, $i \to d$, $s \to d$.

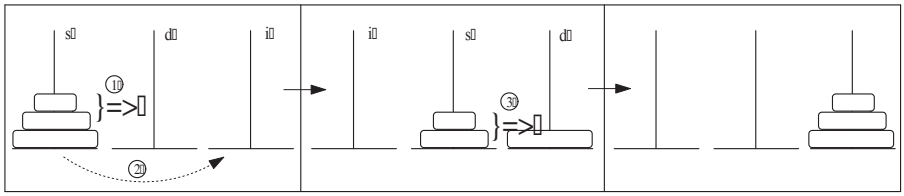

Figura 5.2: Ideea de rezolvare a problemei turnurilor din Hanoi ($n = 3$)

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

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

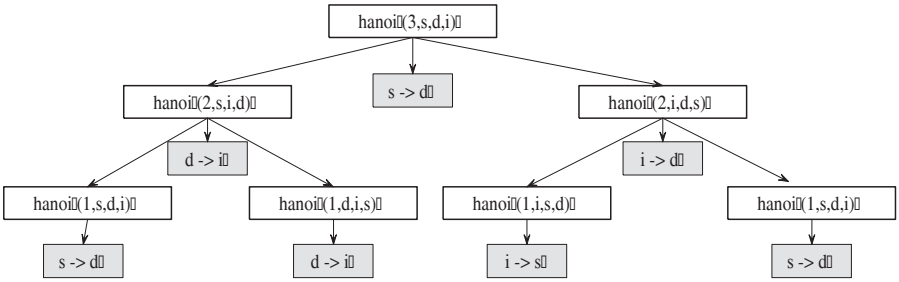

Figura 5.3: Arborele de apel pentru algoritmul `hanoi` când $n = 3$

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 $\Theta(2^n)$.

5.4.3 Căutare binară

Să considerăm problema căutării unei valori v într-un tablou $a[1..n]$ ordonat crescător. Tehnica reducerii 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 se continuă în subtabloul $a[j+1..n]$.

Alegerea cea mai naturală pentru j este să fie cât mai aproape de mijlocul tabloului, astfel dimensiunea problemei este redusă prin împărţire la 2. Un astfel de proces de căutare este cunoscut sub numele de *căutare binară* şi o primă variantă de implementare este descrisă în Algoritmul 5.8 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 $d = n$.

Variante iterative ale tehnicii căutării binare sunt descrise în Algoritmul 5.9. Varianta `cautbin3` evită specificarea compararea de două ori în aceeaşi

Algoritmul 5.8 Variantă recursivă a algoritmului de căutare binară

```
cautbin1(a[s..d],v)
if s > d then
  return false
else
  m ← ⌊(s + d)/2⌋
  if a[m] = v then
    return true
  else if v < a[m] then
    return cautbin1(a[s..m − 1], v)
  else
    return cautbin1(a[m + 1..d], v)
  end if
end if
```

iteraţie a valorii căutate cu elemente ale tabloului ($a[m] = v$ şi $v < a[m]$) şi este mai uşor de analizat însă nu conduce neapărat la un număr mai mic de comparaţii.

Algoritmul 5.9 Variante iterative ale algoritmului de căutare binară

```
cautbin2(a[1..n],v)
s ← 1
d ← n
gasit ← false
while (s ≤ d) and (gasit =false) do
  m ← ⌊(s + d)/2⌋
  if a[m] = v then
    gasit ← true
  else if v < a[m] then
    d ← m − 1
  else
    s ← m + 1
  end if
end while
return gasit
```

```
cautbin3(a[1..n],v)
s ← 1
d ← n
while s < d do
  m ← ⌊(s + d)/2⌋
  if v ≤ a[m] then
    d ← m
  else
    s ← m + 1
  end if
end while
if v = a[s] then
  return true
else
  return false
end if
```

Pentru a analiza algoritmul căutării binare să considerăm că $T(n)$ reprezintă numărul maxim de comparaţii (la fiecare etapă se contorizează o singură comparaţie) 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 $n = 2^m$. Aplicând metoda iteraţiei obţinem:

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

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 $n \in \mathbb{N}$ şi $\lfloor x \rfloor + 1 = \lceil x \rceil$ pentru $x \notin \mathbb{N}$. Prin urmare căutarea binară este de complexitate $\mathcal{O}(\lg\ n)$. Sigur că rezultatul pentru n arbitrar poate fi obţinut aplicând direct Propoziţia 5.1.

5.5 Probleme

Problema 5.1 (*Permutări în ordine lexicografică*) Să se genereze toate permutările de ordin n în ordine lexicografică (în ordinea crescătoare a valorii asociate permutării). Pentru $n = 3$ aceasta înseamnă generarea valorilor în ordinea: $(1, 2, 3)$, $(1, 3, 2)$, $(2, 1, 3)$, $(2, 3, 1)$, $(3, 1, 2)$, $(3, 2, 1)$.

Rezolvare. Se porneşte de la permutarea identică $p[1..n]$, $p[i] = i, i = \overline{1, n}$ (care are asociată valoarea $123 \ldots n$) şi se generează succesiv valoarea imediat următoare constituită din aceleaşi cifre. Pentru fiecare nouă permutare generată se parcurg etapele:

- identifică cel mai mare indice $i \in \{2, 3, \ldots, n\}$ cu proprietatea $p[i] > p[i - 1]$;