31. COMPLEXITATEA STRUCTURILOR DE DATE

31.1 Noţiunea de complexitate

În practică apare adeseori necesitatea comparării diverşilor algoritmi existenţi pentru rezolvarea unei anumite probleme în funcţie de criterii de performanţă. Pentru evaluarea algoritmilor care lucrează cu structuri de date există două tipuri de criterii: statice şi dinamice. Criteriile statice sunt în general subiective şi se referă la aspecte precum eleganţa şi simplitatea algoritmului sau calitatea codului. Singurul criteriu static esenţial pentru un algoritm este cel al corectitudinii, nerespectarea acestuia făcând inutilă orice analiză ulterioară. Criteriile dinamice se referă la factori de comportament. Cele mai importante criterii sunt cele legate de consumul de resurse: timp de execuţie, spaţiu de memorie şi volumul operaţiilor de I/E.

În lucrul cu structuri de date pentru modelarea proceselor social-economice se folosesc modele complexe care presupun un consum foarte mare de resurse pentru rezolvare. Metodele de evaluare a complexităţii oferă posibilitatea depistării componentelor care presupun un consum foarte mare de timp de execuţie şi permit simplificarea acestora prin reducerea dimensiunii problemei sau a complexităţii calculelor (de exemplu prin eliminarea elementelor neliniare şi acceptarea unei pierderi de precizie).

Prezentul capitol prezintă fundamentele teoretice ale metodelor folosite pentru analiza complexităţii algoritmilor. În cadrul capitolului sunt prezentate modalităţile de evaluare a algoritmilor în funcţie de criteriile de performanţă, precum şi aparatul matematic necesar. Pentru simplitate ne vom referi în continuare numai la aspectele legate de timpul de execuţie; cuantificarea utilizării celorlalte tipuri de resurse se poate face prin analogie.

Pentru a putea compara algoritmii avem nevoie în primul rând de o metodologie de măsurare. O primă modalitate ar fi implementarea algoritmilor şi măsurarea timpului necesar execuţiei folosind o baterie de date de test. Deşi foarte precisă, această abordare are numeroase dezavantaje: necesită implementarea algoritmului, depinde de maşina pe care se execută şi de datele de test şi nu oferă nici o fundamentare formală pentru determinarea comportamentului algoritmului pe alte date de test. Pentru a rezolva aceste probleme putem să folosim o abordare mai riguroasă: construim un model al tehnologiei de implementare folosită în care includem resursele utilizate şi costul acestora, după care analizăm comportarea dinamică a algoritmului folosind acest model. Chiar şi analiza unui algoritm simplu folosind o astfel de abordare necesită utilizarea unor mecanisme matematice complicate şi un proces laborios de calcul. De asemenea, datorită faptului ca un algoritm se poate comporta diferit pentru diferite date de intrare, vom avea nevoie şi de un mecanism de a consolida informaţiile obţinute din analiză în formule simple, folosibile în practică.

Pentru eliminarea acestor dificultăţi a fost propusă o abordare bazată pe noţiunea de complexitate asimptotică pentru evaluarea performanţelor algoritmilor. Spunem că un algoritm este cu atât mai complex cu cât consumul de resurse este mai mare. Analiza complexităţii îşi propune să determine clasa de complexitate a algoritmului în funcţie de doi parametri esenţiali: operaţiile critice efectuate de algoritm şi talia problemei de rezolvat.

Operaţiile critice (sau elementare) sunt stabilite în funcţie de specificul problemei de rezolvat; ele sunt acele operaţii care prin natura lor sau prin faptul că sunt executate frecvent în cursul execuţiei programului implică un consum mare de resurse. De exemplu, în cazul algoritmilor de sortare, operaţia critică este compararea a două chei. Pentru un algoritm de căutare într-o bază de date operaţia critică poate fi reprezentă de o citire de pe disc. Prelucrările considerate operaţii elementare nu trebuie sa fie neapărat omogene. Putem considera de exemplu ca orice operaţie aritmetico-logică este o operaţie elementară. De asemenea putem considera un grup de operaţii ca fiind o operaţie elementară.

Talia problemei este definită ca acel parametru sau acei parametri de care depinde frecvenţa de execuţie a operaţiilor critice. Modalităţile de exprimare ale acesteia sunt foarte variate. Astfel, pentru algoritmii de sortare sau pentru algoritmii de căutare cea mai naturală modalitate de exprimare este numărul de elemente ale şirului de intrare. Pentru multe alte probleme, cum ar fi înmulţirea a doi întregi sau verificarea primalităţii unui număr, cea mai bună măsură a taliei problemei este numărul total de biţi necesar pentru reprezentarea numerelor în notaţia binară. Uneori este mai indicat să descriem dimensiunea problemei de rezolvat in funcţie de mai mulţi parametri. De exemplu, în cazul algoritmilor care fac prelucrări pe grafuri, talia problemei este determinată de numărul de noduri şi numărul de arce.

Pentru a putea compara algoritmii din punctul de vedere al performanţelor avem nevoie de o metrică a complexităţii. Aceasta trebuie să permită atât o determinare corectă a resurselor folosite cât şi o simplificare a procesului analiză. În majoritatea cazurilor, analiza exactă a complexităţii este dificilă şi chiar inutilă. Nu avem nevoie de o valoare exactă a volumului de resurse consumate de algoritm, ci doar de o estimare a acesteia astfel încât algoritmul să poată fi încadrat într-o clasă de complexitate. Metrica folosită este bazată pe noţiunile de timp de execuţie şi rată de creştere.

Timpul de execuţie al unui algoritm este definit ca numărul de operaţii critice sau "paşi" executaţi. Pentru simplitate se poate presupune că toate operaţiile se execută în timp constant. Vom vedea mai departe că această simplificare nu influenţează rezultatul analizei. Singura restricţie impusă în alegerea operaţiei în funcţie de care se determină timpul de execuţie este aceea de a nu depinde de dimensiunea problemei. Nu putem considera un grup de *n* comparaţii ca o operaţie elementară în cazul unui algoritm care sortează un sir de *n* numere naturale. Timpul de execuţie este exprimat în general ca o funcţie de talia problemei. Pot exista cazuri în care timpul de execuţie depinde şi de valorile efective ale datelor de intrare, nu numai de volumul acestora. În aceste cazuri se vor determina, în funcţie de specificul problemei, timpii de execuţie în cazul cel mai defavorabil şi eventual pentru cazul mediu.

Deoarece în majoritatea cazurilor timpul de execuţie este dificil de determinat, în analiza complexităţii metrica folosită pentru clasificarea algoritmilor este rata de creştere a timpului de execuţie. Aceasta măsoară valoarea asimptotică (când talia problemei tinde către infinit) a performanţei. Pentru determinarea comportamentului vom lua in considerare numai termenul dominant al funcţiei care reprezintă timpul de execuţie, ignorând coeficientul acestuia şi termenii de rang inferior. Deşi pentru date de intrare de talie mică această metrică poate conduce la

rezultate eronate, pentru valori suficient de mari permite o caracterizare corectă şi o clasificare consistentă a performanţelor algoritmilor.

31.2 Modelarea matematică a complexităţii algoritmilor

Noţiunilor intuitive prezentate in secţiunea anterioară le-a fost asociat un set de notaţii matematice. În continuare vom prezenta definiţiile matematice ale conceptelor utilizate, precum şi proprietăţile importante ale acestora.

Presupunem că funcţiile care descriu cantitativ timpul de execuţie al algoritmului sunt de variabilă întreagă, cu valori reale pozitive, deci de forma $f : N \to R_+$. Parametrul funcţiei reprezintă talia problemei, iar valoarea funcţiei reprezintă numărul de operaţii elementare necesare.

Pentru o funcţie dată $g(n)$ definim clasa $\Theta(g(n))$ a algoritmilor ca totalitatea algoritmilor cu de timpi de execuţie caracterizaţi de o funcţie *f*(*n*) din mulţimea:

$$\Theta(g(n)) = \{ f(n) \mid (\exists c_1, c_2 \in R_+)(\exists n_0 \in N)(\forall n \geq n_0)[0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)] \}$$

O funcţie $f(n)$ aparţine mulţimii $\Theta(g(n))$ dacă există două constante pozitive c_1 şi c_2 astfel încât, pentru un n suficient de mare, este încadrată de $c_1 g(n)$ şi $c_2 g(n)$. Deşi Θ este o mulţime, apartenenţa se notează cu $f(n) = \Theta(g(n))$ şi se citeşte "*f* este în clasa $\Theta(g(n))$" sau "*f* are ordinul *g*(*n*)". În aceste condiţii spunem ca *g*(*n*) este o limită asimptotică "strânsă" pentru *f*(*n*).

Notaţia Θ este utilizabilă numai atunci când funcţia care caracterizează timpul de execuţie este unică. Pentru situaţiile în care nu se poate determina o astfel de funcţie se foloseşte notaţia O. Pentru o funcţie dată $g(n)$, definim clasa $O(g(n))$ a algoritmilor ca totalitatea algoritmilor cu timpi de execuţie caracterizaţi de o funcţie *f*(*n*) din mulţimea $O(g(n)) = \{ f(n) \mid (\exists c \in R_+)(\exists n_0 \in N)(\forall n \geq n_0)[0 \leq f(n) \leq cg(n)] \}$.

Spre deosebire de Θ, notaţia O oferă doar o limită asimptotică superioară, fără a face nici o precizare despre cât de strânsă este această limită. Aceste caracteristici ale notaţiei o fac potrivită pentru analiza comportării algoritmilor în cazul cel mai defavorabil. Notaţia stabileşte o limită superioară a timpului de execuţie independentă de structura datelor de intrare. Dacă timpul de execuţie al unui algoritm este de ordin $O(g(n))$, atunci, indiferent de valorile datelor de intrare de dimensiune *n*, timpul de execuţie este cel mult egal cu *cg*(*n*). Similar se poate defini si o notaţie pentru determinarea limitei asimptotice inferioare.

Pentru o funcţie dată $g(n)$, clasa $\Omega(g(n))$ a algoritmilor este formată din totalitatea algoritmilor cu de timpi de execuţie caracterizaţi de o funcţie *f*(*n*) din mulţimea:

$$\Omega(g(n)) = \{ f(n) \mid (\exists c \in R_+)(\exists n_0 \in N)(\forall n \geq n_0)[0 \leq cg(n) \leq f(n)] \}.$$

Această notaţie este folosită în special pentru a caracteriza comportarea algoritmului în cazul cel mai favorabil.

Figura 31.1 prezintă interpretarea grafică a celor trei notaţii.

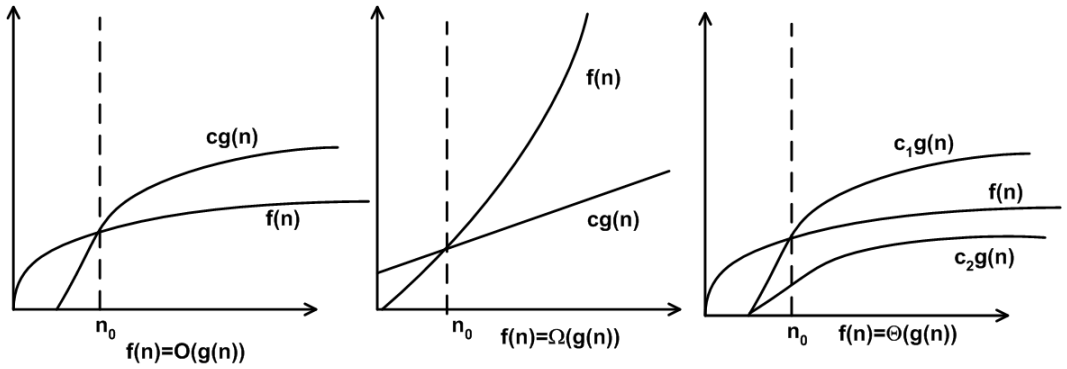

Figura 31.1 *interpretarea grafică a notaţiilor de complexitate*

Se poate demonstra uşor că între notaţiile de complexitate există următoarele relaţii:

$$\left[\left(f(n)=O(g(n))\right)\wedge\left(f(n)=\Omega(g(n))\right)\right]\Leftrightarrow\left(f(n)=\Theta(g(n)\right) \qquad (31.1)$$

relaţia de dualitate: $\left(f(n)=O(g(n))\right)\Leftrightarrow\left(g(n)=\Omega(f(n))\right)$ *(31.2)*

Pentru a desemna limite inferioare sau superioare care nu sunt asimptotic strânse se utilizează notaţiile *o* si ω, care sunt definite similar notaţiilor O, respectiv Ω, astfel:

$$o\left(g(n)\right)=\left\{f(n)\,|\,(\forall c\in R_+)(\exists n_0\in N)(\forall n\geq n_0)\,[0\leq f(n)\leq cg(n)]\right\} \qquad (31.3)$$

respectiv:

$$\omega\left(g(n)\right)=\left\{f(n)\,|\,(\forall c\in R_+)(\exists n_0\in N)(\forall n\geq n_0)\,[0\leq cg(n)\leq f(n)]\right\} \qquad (31.4)$$

Diferenţa faţă de notaţiile principale este că inegalitatea este verificată pentru orice constantă *c* pozitivă, ceea ce implică intuitiv că funcţia *f*(*n*) devine nesemnificativă în raport cu *g*(*n*) pentru *n* foarte mare (în cazul notaţiei *o*), adică $\lim_{n\to\infty}\frac{f(n)}{g(n)}=0$.

În cazul în care funcţia caracteristică a unui algoritm depinde de mai multe variabile (exemplu: numărul de noduri şi numărul de arce în cazul unui graf) se pot analiza separat componentele folosind notaţiile prezentate sau se pot extinde notaţiile pentru a lucra cu funcţii de tipul $N\times N\to N\times N$.

31.3 Clase de complexitate

Se poate demonstra uşor că notaţiile de complexitate au următoarele proprietăţi relaţionale pe mulţimea funcţiilor $N\to R_+$:

Tranzitivitate:

- $f(n)=\Theta(g(n))$ şi $g(n)=\Theta(h(n))$ implică $g(n)=\Theta(h(n))$;
- $f(n)=O(g(n))$ şi $g(n)=O(h(n))$ implică $g(n)=O(h(n))$;
- $f(n)=\Omega(g(n))$ şi $g(n)=\Omega(h(n))$ implică $g(n)=\Omega(h(n))$;

- $f(n) = o(g(n))$ şi $g(n) = o(h(n))$ implică $g(n) = o(h(n))$;
- $f(n) = \omega(g(n))$ şi $g(n) = \omega(h(n))$ implică $g(n) = \omega(h(n))$.

Reflexivitate:
- $f(n) = \Theta(f(n))$;
- $f(n) = O(f(n))$;
- $f(n) = \Omega(f(n))$.

Simetrie: $f(n) = \Theta(g(n))$ dacă şi numai dacă $g(n) = \Theta(f(n))$.

Antisimetrie:
- $f(n) = O(g(n))$ dacă şi numai dacă $g(n) = \Omega(f(n))$;
- $f(n) = o(g(n))$ dacă şi numai dacă $g(n) = \omega(f(n))$.

Notaţiile O şi Ω, care sunt tranzitive, antisimetrice si tranzitive, formează relaţii de ordine parţială peste mulţimea funcţiilor $N \to R_+$, iar Θ, care este reflexivă, simetrică si tranzitivă, formează o relaţie de echivalenţă. Aceste relaţii pot fi asimilate cu relaţiile de ordine de pe mulţimea numerelor reale astfel: $f(n) = \Theta(g(n)) \approx a = b$, $f(n) = O(g(n)) \approx a \le b$, $f(n) = \Omega(g(n)) \approx a \ge b$, $f(n) = o(g(n)) \approx a < b$ şi $f(n) = \omega(g(n)) \approx a > b$. O singură proprietate a numerelor reale nu este valabilă şi pentru notaţiile de complexitate trihotomia (pentru oricare numere reale *a* şi *b* exact una dintre relaţiile *a<b, a>b* sau *a=b* este adevărată). Aceasta arată ca există funcţii care nu pot fi comparate asimptotic (de exemplu *n* si $n^{1+\sin(n)}$).

Faptul că Θ, O şi Ω pot fi văzute ca relaţii de ordine, chiar şi parţiale, ajută la compararea algoritmilor caracterizaţi de funcţii din clasele respective de complexitate. De asemenea, acestea pot fi folosite pentru a construi clase de performanţă pentru algoritmi. Tabelul 31.1 prezintă principalele clase de complexitate ale algoritmilor care apar în lucrul cu structuri de date.

Tabelul nr. 31.1 *Clase de complexitate*

Clasa	**Exemple**	**Număr de operaţii**		
		n=3	**n=10**	**n=100**
$O(1)$	Operaţii aritmetice simple, căutare folosind tabele de dispersie	1	1	1
$O(\ln(n))$	Căutare binară	$\cong$1	$\cong$2	$\cong$5
$O(n)$	Căutare secvenţială, suma unui şir	3	10	100
$O(n\ln(n))$	Sortările rapide (interclasare, sortare rapidă, ...)	$\cong$3	$\cong$23	$\cong$460
$O(n^2)$	Sortările "naive"(metoda bulelor, inserţie, ...), parcurgeri de matrice pătratice de ordin *n*	9	100	10000
$O(n^3)$	Înmulţirea clasică a matricelor	27	1000	1000000
$O(2^n)$	Generare aranjamente	8	1024	$\cong 10^{30}$

Pentru o problemă dată, căutăm mereu sa găsim un algoritm care să se situeze într-o clasă de complexitate cat mai mică. După cum se observă şi din numărul de operaţii necesar diferitelor clase de complexitate, in cazul

in care lucrăm cu un volum mare de date clasa algoritmului devine foarte importantă.

Pentru volume mari de date, diferenţele dintre diferite maşini de calcul şi dintre compilatoare devin neesenţiale în comparaţie cu diferenţele dintre clasele de algoritmi. Pentru edificare vom considera următoarele implementări ale unei soluţii informatice destinate sortării unui volum mare de înregistrări:

- implementare a algoritmului de sortare prin inserţie ($O(n^2)$) pe un supercalculator capabil sa execute 100 de milioane de instrucţiuni pe secundă; presupunem că implementarea a fost făcută şi optimizată folosind un limbaj de asamblare, obţinându-se astfel un timp de execuţie $2n^2$;
- implementare a algoritmului de sortare prin interclasare ($O(n\lg(n))$) pe un calculator personal capabil sa execute numai 1 milion de instrucţiuni pe secundă; presupunem că implementarea a fost făcută folosind un limbaj de nivel înalt cu un compilator ineficient, obţinându-se astfel un timp de execuţie de $50n\lg n$.

Pentru a sorta o serie de un milion de înregistrări vor fi necesari următorii timpi:

- $T = \frac{2 \cdot \left(10^6\right)^2 \text{ instructiuni}}{10^8 \text{ instructiuni/secundă}} = 20000 \text{ secunde} \cong 6 \text{ ore}$ pentru un super calculator;
- $T = \frac{2 \cdot 10^6 \cdot \lg(10^6) \text{ instructiuni}}{10^6 \text{ instructiuni / secunda}} = 1000 \text{ secunde} \cong 17 \text{ minute}$ pentru un PC.

Se observă că, folosind un algoritm dintr-o clasă de complexitate mai mică, calculatorul personal, chiar si folosind un compilator ineficient, a fost de 20 de ori mai rapid decât supercalculatorul. Acest exemplu arată că alegerea algoritmului este determinantă pentru performanţele sistemelor. Optimizarea implementării şi alegerea echipamentelor hardware devin importante numai după ce a fost ales un algoritm eficient.

31.4 Metode de calcul pentru indicatorii de complexitate

Nu există nici o formulă generală pentru determinarea complexităţii. Aceasta se face de la caz la caz, ţinând cont de particularităţile problemei. În general se urmează următorii paşi:

- determinarea parametrilor de care depinde talia problemei (aceştia vor fi parametrii funcţiei care caracterizează timpul de execuţie şi implicit ai funcţiei caracteristice a clasei de complexitate in care va fi încadrat algoritmul);
- alegerea operaţiilor elementare (această alegere va determina valoarea funcţiei pentru o valoare a parametrilor);
- determinarea formulei funcţiei care caracterizează timpul de execuţie ;
- încadrarea algoritmului într-o clasă de complexitate şi/sau compararea lui cu alţi algoritmi care rezolvă aceeaşi problemă.

În legătură cu determinarea funcţiei caracteristice trebuie făcute unele precizări:

- nu este necesară determinarea precisă a acesteia; este suficient să se determine numai termenul dominant al acesteia;
- în cazul în care depinde şi de valorile particulare ale datelor de intrare (nu numai de volumul acestora) se vor determina funcţiile in cazul cel mai favorabil şi în cazul cel mai defavorabil pentru a se stabili limitele asimptotice şi, dacă se consideră necesar, se vor determina funcţii şi pentru valori intermediare pentru stabilirea unui comportament mediu.

Metode generale

Pentru calculul funcţiei caracteristice si pentru încadrarea acesteia într-o clasă de complexitate putem folosi următoarele proprietăţi de bază ale notaţiilor:

Compunerea:

- dacă avem un algoritm compus din două secvenţe cu timpii de execuţie caracterizaţi de funcţiile *f*(*n*), respectiv *g*(*n*), atunci algoritmul va fi de ordinul *max(f,g)*: $\Theta(f+g)=\Theta(\max(f,g))$; relaţia este valabilă şi pentru celelalte notaţii; se poate generaliza ca $\Theta\left(\sum_{i=1}^{k} f_k(n)\right)=\Theta\left(\max_{k=1,k}(f_k(n))\right)$, *k* – constantă;
- dacă un algoritm apelează o secvenţă de ordin *f*(*n*) de *k* ori (*k* – constantă), atunci algoritmul va avea tot ordinul *f*(*n*): $\Theta(kf(n))=\Theta(f(n))$;
- dacă un algoritm apelează o secvenţă de ordin *f*(*n*) intr-un ciclu care se execută de *g*(*n*) ori, atunci algoritmul va avea ordinul $\Theta(g(n)\cdot f(n))$.

Utilizarea limitelor pentru determinarea apartenenţei la o clasă (implicaţiile inverse sunt adevărate, dar numai în cazul în care există limitele):

- dacă $\lim_{n\to\infty}\frac{f(n)}{g(n)}=0$, atunci $f(n)=o(g(n))$ ($f(n)=O(g(n))$ şi $f(n)\neq\Theta(n)$)
- dacă $\lim_{n\to\infty}\frac{f(n)}{g(n)}\in R_+$, atunci $f(n)=\Theta(g(n))$
- dacă $\lim_{n\to\infty}\frac{f(n)}{g(n)}=+\infty$, atunci $f(n)=\omega(g(n))$ ($f(n)=\Omega(g(n))$ şi $f(n)\neq\Theta(n)$)

Folosind limitele se pot demonstra imediat următoarele proprietăţi:

- complexitatea funcţiilor polinomiale: dacă $f(n)=\sum_{i=0}^{m}a_i n^i$, atunci $f(n)=\Theta(n^m)$;
- pentru orice constante reale *a* şi *b*, *b*>0 avem $(n+a)^b=\Theta(n^b)$;
- pentru orice constantă reală *a*>1, dacă $f(n)=\Theta(g(n))$ avem $a^{f(n)}=\Theta(a^{g(n)})$.

Proprietăţile sumelor şi produselor:

- din proprietatea de liniaritate a seriilor avem $\sum_{k=1}^{n}\Theta(f(k))=\Theta\left(\sum_{i=1}^{n}f(k)\right)$;
- serii aritmetice: $\sum_{k=1}^{n}k=\frac{n(n+1)}{2}=\Theta(n^2)$;
- serii geometrice: $\sum_{k=1}^{n}x^k=\frac{x^{n+1}-1}{x-1}=\Theta(x^n)$, $x>1$;
- serii armonice: $\sum_{k=1}^{n}\frac{1}{k}=\ln(n)+\Theta(1)$;
- transformarea produselor in sumă: $\ln\left(\prod_{k=1}^{n}a_k\right)=\sum_{k=1}^{n}\ln(a_k)$ şi $\Theta(f)=\Theta(g)\Leftrightarrow\Theta(\ln(f))=\Theta(\ln(g))$.

Pentru sume complexe se pot utiliza tehnici precum inducţia matematică, mărginirea termenilor, despărţirea sumelor sau aproximarea prin integrale.

Metode pentru funcţii recursive

Fie *T*(*n*) funcţia care caracterizează timpul de execuţie al unui algoritm. În multe situaţii avem de-a face cu funcţii recursive. În secţiunile următoare vom prezenta trei tehnici de rezolvare a recurenţelor de complexitate: metoda iteraţiei, metoda substituţiei şi metoda Master. Aceste tehnici pornesc de la observaţia că nu este importantă forma exactă a funcţiei ci doar clasa de complexitate din care face parte. De asemenea putem omite condiţiile de oprire ale recurenţei deoarece putem presupune ca $T(n)=\Theta(1)$ pentru un *n* suficient de mic.

Metoda substituţiei. Metoda presupune "ghicirea" formei soluţiei şi după aceea folosirea inducţiei matematice pentru a găsi constantele şi a demonstra că soluţia este corectă. Numele metodei vine de la substituirea răspunsului ghicit atunci când ipoteza de inducţie este aplicată valorilor mici. Metoda este foarte puternică, dar este aplicabilă doar in cazurile in care se poate intui forma soluţiei.

Pentru exemplificare vom analiza funcţia $T(n)=\begin{cases}T(\lfloor n/2\rfloor) & ,n>1\\ 1 & ,n=1\end{cases}$ corespunzătoare unei căutări binare, pentru care presupunem că ordinul de complexitate va fi $O(\log n)$. Pentru a demonstra această afirmaţie vom folosi inducţia matematică.

Presupunem ca avem $T(\lfloor n/2\rfloor)\le c\log(n/2)$ pentru o constantă $c>0$ aleasă convenabil. Vom demonstra că afirmaţia este valabilă şi pentru $T(n)$. Avem $T(n)=T(\lfloor n/2\rfloor)+1\le c\log(n/2)+1=c\left(\log n+\log 2^{-1}\right)+1=c\log n+(1-c)$, de unde obţinem $T(n)\le c\log n$ pentru $c\ge 1$. Pentru ca demonstraţia să fie completă trebuie să arătam că este respectată condiţia şi pentru cazul iniţial: $T(n_0)\le c\log n_0$. Se observă că nu este necesară demonstrarea afirmaţiei pentru cazul limită $n=1$, deoarece, din definiţia complexităţii, cunoaştem faptul că este suficient ca afirmaţia să fie validă pentru toţi $n>n_0$, unde n_0 este o constantă pozitivă oarecare. În cazul de faţă, pentru

$n_0 = 2$ avem $T(2) = 2 \le c \log 2$ pentru oricare $c > 2$. In concluzie avem $T(n) \le c \log n$ pentru oricare $n \ge n_0 = 2$ şi oricare constantă $c \ge 2$.

Procesul de demonstrare a unei recurenţe de complexitate folosind metoda substituţiei are următorii paşi:

- intuirea unei soluţii $f(n)$;
- verificarea relaţiei $T(n) \le cf(n)$ prin substituirea lui $T(n)$ cu $f(n)$;
- verificarea cazului limită $T(n_0) \le cf(n_0)$.

Metoda substituţiei poate fi folosită atât pentru determinarea unei limite superioare, cât şi pentru determinarea unei limite inferioare sau a unei încadrări exacte a timpului de execuţie al algoritmului (caz în care inegalităţile vor fi substituite cu egalităţi). În cazul în care expresia este prea complexă şi nu se poate intui uşor o soluţie se pot demonstra limite superioare şi inferioare largi pentru algoritm, care vor fi apoi restrânse progresiv.

Chiar şi atunci când este găsită o limită asimptotică bună, pot apărea probleme la demonstrarea prin inducţie. De obicei, problema este considerarea unei ipoteze de inducţie prea slabe şi poate fi rezolvată prin scăderea unor termeni de rang inferior. De exemplu, pentru recurenţa $T(n) = 2T(n/2) + 1$ încercăm să arătăm că $T(n) = O(n)$ demonstrând că $T(n) \le cn$. Vom avea $T(n) \le 2c(n/2) + 1 = cn + 1$, care nu implică $T(n) \le cn$ pentru oricare *c*. Pentru depăşirea acestei probleme, vom considera o ipoteză de inducţie mai puternică: $T(n) \le cn - b$, unde *b* este o constantă pozitivă. Folosindu-ne de această ipoteză obţinem: $T(n) \le 2(c\frac{n}{2} - b) + 1 = cn - 2b + 1 \le cn - b$ pentru $b \ge 1$. Principiul aplicat este acela că pentru a demonstra o condiţie mai strânsă pentru valori mari trebuie să pornim de la ipoteze mai puternice.

O altă posibilitate de a simplifica recurenţele în vederea aplicării metodei este schimbarea de variabilă. De exemplu, în recurenţa $T(n) = 2T(\sqrt{n}) + \log n$ făcând schimbarea de variabilă $m = \log n$ obţinem recurenţa echivalentă $S(m) = 2S(m/2) + m$ care se poate rezolva uşor, obţinându-se soluţia $S(m) = O(m \log m)$, de unde rezultă $T(n) = O(\log n \cdot \log(\log n))$.

Metoda iteraţiei

Această metoda transformă recurenţa într-o sumă şi se bazează pe folosirea tehnicilor de mărginire a sumelor pentru rezolvare. Conversia recurenţei în sumă se face prin expandarea recurenţei şi exprimarea ca o sumă de termeni dependenţi numai de *n* şi de condiţiile iniţiale.

Pentru exemplificare vom considera problema turnurilor din Hanoi care are un timp de execuţie caracterizat de următoarea recurenţă:

$$T(n) = \begin{cases} T(n-1) + 1 + T(n-1) = 2T(n-1) & ,n > 1 \\ 1 & ,n = 1 \end{cases} \qquad (31.5)$$

Prin expandarea expresiei se obţine:

$$T(n) = 2T(n-1) + 1 = 2\big(2T(n-2)+1\big) + 1 = 4T(n-1) + 2 + 1 =$$
$$= 4\big(2T(n-2)+1\big) + 2 + 1 = 8T(n-2) + 4 + 2 + 1 = \ldots = \qquad (31.6)$$
$$= \sum_{i=0}^{n-1} 2^i$$

care este o progresie geometrică cu soluţia $2^n - 1$, deci avem $T(n) = O(2^n)$.

În general, aplicarea metodei conduce la calcule algebrice complicate si necesită o atenţie sporită la exactitatea relaţiilor. Punctele cheie care trebuie urmărite în analiză sunt numărul de iteraţii necesare pentru atingerea condiţiilor limită şi suma de termeni existentă pe fiecare nivel. Uneori, în procesul de iterare, soluţia poate fi "văzută" fără a efectua toate calculele; în acest caz se poate abandona metoda şi se va recurge la metoda substituţiei care este mai puţin laborioasă.

O problemă mai deosebită în aplicarea metodei care poate apărea în cazul în care funcţia care trebuie analizată conţine funcţiile parte întreagă inferioară sau superioară ($\lfloor\ \rfloor$ sau $\lceil\ \rceil$). În general, aceste cazuri pot fi rezolvate uşor numai dacă se iau în considerare numai puterile exacte ale lui *n*. Pentru exemplificare vom considera următoarea recurenţă corespunzătoare unei sortări prin interclasare:

$$f(n) = \begin{cases} T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n & ,n > 1 \\ 1 & ,n = 1 \end{cases} \qquad (31.7)$$

Pentru rezolvarea recurenţelor de acest tip putem folosi următoarea proprietate: Dacă o funcţie *f* este netedă ($\exists b \geq 2$ a.î. $f(bn) = O(f(n))$), $a \geq 2$ şi există un întreg oarecare şi *t* o funcţie nedescrescătoare a.î. $t(n) = O(f)$ când *n* este o putere exactă a lui *a*, atunci $t(n) = O(f)$ pentru oricare *n*.

Vom considera pentru exemplul prezentat că *n* este o putere exactă a lui 2 şi obţinem:

$$T(n) = 2T(n/2) + n = 2\big(T(n/4) + n/2\big) + n = 2^2 T\left(\frac{1}{2^2}\right) + 2n = \ldots \qquad (31.8)$$
$$= 2^{\log_2 n} + \log_n(n) \cdot n = n + \log_n(n) \cdot n$$

deci $T(n)=O(n\ log(n))$ pentru *n* putere exactă a lui 2. Aplicând proprietatea anterioară obţinem $T(n)=O(n\ log(n))$ pentru oricare *n*.

Metoda Master

Metoda Master se bazează pe teorema cu acelaşi nume şi oferă o soluţie directă pentru recurenţele de forma $T(n) = aT(n/b) + f(n)$, unde $a \geq 1$ şi $b > 1$ sunt constante, iar $f(n)$ este o funcţie asimptotic pozitivă. Această recurenţă caracterizează timpul de execuţie al unui algoritm care divide problema in *a* sub-probleme de dimensiune *n*/*b* pe care le rezolvă recursiv, iar costul necesar divizării problemei si combinării soluţiei este $f(n)$. Pentru corectitudine termenul $T(n/b)$ trebuie scris ca $T(\lfloor n/b \rfloor)$ sau $T(\lceil n/b \rceil)$ (deoarece funcţia T este definită pe mulţimea numerelor naturale), dar acest lucru nu afectează rezultatul teoremei.

Teorema Master: Dacă $a \geq 1$ şi $b > 1$ sunt constante, $f(n)$ este o funcţie asimptotic pozitivă şi $T(n)$ este o funcţie nenegativă definită de

recurenţa $T(n) = aT(n/b) + f(n)$ (unde *a*/*b* este interpretat ca $T(\lfloor n/b \rfloor)$ sau $T(\lceil n/b \rceil)$), atunci *T*(*n*) poate fi mărginit asimptotic astfel:

1. dacă $f(n) = O(n^{\log_b a - \varepsilon})$ pentru o constantă $\varepsilon > 0$, atunci $T(n) = \Theta(n^{\log_b a})$;
2. dacă $f(n) = \Theta(n^{\log_b a})$, atunci $T(n) = \Theta(n^{\log_b a} \cdot \lg(n))$;
3. dacă $f(n) = \Omega(n^{\log_b a + \varepsilon})$ pentru o constantă $\varepsilon > 0$ şi $a \cdot f(n/b) \le cf(n)$ pentru o constantă $c < 1$ şi oricare n suficient de mare, atunci $T(n) = \Theta(f(n))$.

După cum rezultă din teoremă, nu orice recurenţă poate fi rezolvată folosind această metodă. Uneori teorema nu poate fi aplicată direct, iar alteori nu poate fi aplicată deloc, dar, atunci când este aplicabilă, ea înlocuieşte un calcul greoi cu verificarea unor condiţii între parametrii recurenţei.

Pentru exemplificare vom considera recurenţa $T(n) = 2T\left(\sqrt{n}\right) + \log n$. Observăm că teorema nu se poate aplica direct şi facem următoarea transformare (folosind relaţia $a^{\log_b n} = n^{\log_b a}$ cu *a*=*b*=2): $T(n) = T(2^{\log_2 n}) = 2T(2^{(\log n)/2}) + 2^{\log_2 n}$, unde, făcând substituţia $S(m) = T(2^{\log_2 n})$, obţinem recurenţa echivalentă $S(m) = 2S(m/2) + m$. În acest caz putem aplica teorema (cazul 2) cu *a*=*b*=2, *f*(*n*)=Θ(*n*) şi $n^{\log_a b} = n^{\log_2 2} = n$ şi obţinem $S(n) = \Theta\left(n^{\log_a b} \log_2 n\right) = \Theta\left(n \log_2 n\right)$.

Revenind la recurenţa iniţială obţinem:

$$T(n) = T(2^{\log_2 n}) = S(m) = \Theta\left(m \log_2 m\right) = \Theta\left(\log_2 n \log_2 \left(\log_2 n\right)\right) \qquad (31.9)$$