I. ANALIZA ALGORITMILOR

PRELIMINARII

Abu Ja`far Mohammed ibn Musa al-Khowarizmi (autor persan, sec. VIII-IX), a scris o carte de matematică cunoscută în traducere latină că "Algorithmi de numero indorum", iar apoi că "Liber algorithmi", unde "algorithm" provine de la "al-Khowarizmi", ceea ce literal înseamnă "din oraşul Khowarizm". În prezent, acest oraş se numeşte Khiva şi se află în Uzbechistan. Atât al-Khowarizmi, cât şi alţi matematicieni din Evul Mediu, înţelegeau prin *algoritm* o regulă pe baza căreia se efectuau calcule aritmetice. Astfel, în timpul lui Adam Riese (sec. XVI), algoritmii foloseau la: dublări, înjumătăţiri, înmulţiri de numere. Alţi algoritmi apar în lucrările lui Stifer ("Arithmetica integra", Nürnberg, 1544) şi Cardano ("Ars magna sive de reguli algebraicis", Nürnberg, 1545). Chiar şi Leibniz vorbeşte de "algoritmi de înmulţire". Termenul a rămas totuşi multă vreme cu o întrebuinţare destul de restrânsă, chiar şi în domeniul matematicii.

Kronecker (în 1886) şi Dedekind (în 1888) semnează actul de naştere al teoriei funcţiilor recursive. Conceptul de recursivitate devine indisolubil legat de cel de algoritm. Dar abia în deceniile al treilea şi al patrulea ale secolului nostru, teoria recursivităţii şi algoritmilor începe să se constituie ca atare, prin lucrările lui Skolem, Ackermann, Sudan, Gödel, Church, Kleene, Turing, Peter şi alţii.

1. ALGORITMI

Un algoritm constă într-o procedură de calcul bine definită care, într-o succesiune finită de paşi, transformă o mulţime de valori (date de intrare) într-o altă mulţime de valori (date de ieşire sau rezultatul prelucrării).

Un algoritm constituie un mijloc de rezolvare a problemelor de calcul bine definite. Un enunţ corect al unei astfel de probleme descrie, în termeni generali, relaţia intrare/ieşire corespunzătoare algoritmului problemei.

Conceptele de funcţie calculabilă şi algoritm formează obiectele principale de studiu în teoria calculabilităţii. Ele apar în diferite domenii ale informaticii şi matematicii sub diverse grade de abstractizare.

Dezvoltarea unor ştiinţe implică, printre altele, construirea unor algoritmi care să rezolve o gamă cât mai variată de probleme.

Noţiunea de algoritm se întâlneşte încă din clasele elementare relativ la cele patru operaţii fundamentale aplicate numerelor naturale reprezentate în baza zece. Mai târziu, se studiază algoritmul lui Euclid şi algoritmi cu privire la rezolvarea unei ecuaţii, determinarea inversei unei matrice nesingulare etc.

Exemplu 1.1:

Să descriem un algoritm de înmulţire a două numere naturale x, y numit înmulţirea „La Russe".

P1. Se consideră un tablou bidimensional cu două coloane. Pe prima linie se află x în prima coloană şi y în a doua coloană.

P2. Numărul din prima coloană se împarte la 2 (împărţire întreagă), iar cel din a doua coloană se înmulţeşte cu 2.

P3. Pasul P2 se repetă până când în prima coloană se obţine numărul natural 1.

P4. Se vor elimina liniile care conţin în prima coloană un număr par.

P5. Se vor însuma numerele naturale din coloana a doua care corespund liniilor rămase.

Suma obţinută reprezintă produsul xy.

Fie numerele $x = 245$ şi $y = 37$. Conform algoritmului descris se obţine:

DEÎNMULŢIT	ÎNMULŢITOR
245	37
122**	74**
61	148
30**	296**
15	592
7	1184
3	2368
1	4736
xy	9065

Liniile marcate cu (**) se consideră eliminate.

Asupra algoritmului descris se pun o serie de întrebări.

a) Condiţia de oprire

Răspunsul algoritmului constă în furnizarea produsului *xy*. Ultima linie din ciclu este cea care conţine 1 în prima coloană. Odată cu ieşirea din ciclu, algoritmul se încheie.

b) Corectitudinea

Algoritmul calculează corect produsul celor două numere naturale. Se observă că dacă marcăm cu 1 liniile rămase şi cu 0 pe cele eliminate, se obţine numărul *x* scris în ordine inversă în binar ($245_{[10]} = 11110101_{[2]}$). De altfel acest algoritm calculează produsul în binar a două numere naturale.

c) Durata execuţiei (viteza de execuţie)

Acest algoritm are aceeaşi viteză de execuţie ca şi algoritmul standard de înmulţire a două numere naturale (efectuând înmulţirea în binar). Există şi alţi algoritmi cu o viteză mai mare care calculează un astfel de produs.

d) Dimensiunea spaţiului de memorie alocat etc.

Teoria mulţimilor constituie un instrument de descriere simplă şi precisă a conceptului de funcţie. Construirea unui model matematic pentru un astfel de concept este relativ uşor de realizat fiindcă avem de-a face cu un proces static. Descrierea conceptului de algoritm este mai dificilă deoarece construirea unui model matematic pentru un astfel de concept presupune un proces dinamic.

Au existat diverşi matematicieni ca: E. Post, K. Gödel, A.M. Turing, A. Church etc. care, în mod independent, au dat câte o descriere a noţiunii de algoritm.

Clasa tuturor algoritmilor, independent de modul de reprezentare a lor, posedă unele trăsături generale.

Pentru un algoritm trebuie precizat:

a) domeniul algoritmului;

b) descrierea propriu-zisă a algoritmului.

Domeniul algoritmului este o mulţime cel mult numărabilă de elemente asupra cărora acţionează algoritmul respectiv în care se includ şi rezultatele finale.

Descrierea propriu-zisă a algoritmului constă în instrucţiuni care se execută asupra elementelor de intrare prezente în domeniul algoritmului. Fiecare din aceste instrucţiuni pune în evidenţă un număr finit de operaţii elementare.

Iata descrierea algoritmului „La Russe" :

```
RUSSE(A, B)
1.  arrays X, Y    {iniţializare}
2.  X[1] ← A; Y[1] ← B
3.  i ← 1 {se construiesc cele două coloane}
4.  while X[i] > 1 do
5.          X[i+1] ← X[i] div 2 {div reprezintă împărţirea întreagă}
6.          Y[i+1] ← Y[i]+Y[i]
7.          i ← i+1
8.  {adună numerele Y[i] corespunzătoare numerelor X[i] impare}
9.  prod ← 0
10. while i > 0 do
11.             if X[i] este impar
12.         then prod ← prod+Y[i]
13.             i ← i-1
14. return prod
```

Acţiunea unui algoritm este subordonată:

a) unor reguli de operare asociate;

b) agentului de calcul.

Regulile de operare asociate unui algoritm solicită fiecărei instrucţiuni să indice locaţiile unde se află datele de intrare cât şi pe cele în care se depun rezultatele intermediare şi finale.

Agentul de calcul poate fi uman, mecanic, electronic etc. şi are rolul de a executa instrucţiunile prezente în algoritm.

O problemă rezolvabilă algoritmic (problemă pentru care există algoritm care să o rezolve) o gândim ca o funcţie P total definită pe mulţimea tuturor datelor de intrare sau informaţiilor iniţiale DI (cel mult numărabilă), cu valori în mulţimea datelor (informaţiilor) finale DF (cel mult numărabilă). O ***instanţă*** a unei astfel de probleme constă într-un element dat $x \in$ DI care satisface condiţiile impuse de enunţul problemei şi este capabil să conducă la o soluţie a acesteia. Dacă $P(x)\downarrow$ (există) şi $P(x) = y$, se spune că *y* este *soluţia* problemei P pentru intrarea *x*. În caz contrar, se spune că P nu are soluţie pentru intrarea *x*.

În vederea prelucrării pe calculator, un astfel de algoritm este transpus într-un program scris într-un limbaj de programare. Acest program evaluează o funcţie naturală. Dacă domeniul de definiţie al funcţiei constituie ***intrarea*** pentru programul respectiv, atunci valorile funcţiei sunt de fapt rezultatele programului.

Exemplul 1.2.

Să considerăm problema identificării unei chei numerice date într-un tablou numeric unidimensional.

Datele de intrare sau intrarea algoritmului: dimensiunea n a tabloului, tabloul numeric $X = (x[1],..., x[n])$ şi valoarea numerică y (cheia);

Datele de ieşire sau ieşirea algoritmului: poziţia acelui element din tablou care coincide cu cheia dată y sau un mesaj de neidentificare în cazul în care această cheie nu coincide cu nici un element din tablou.

O instanţă a problemei propuse constă într-un tablou numeric dat, fie el $X = (-11,121,1,0,7,-48)$ de dimensiune 6 şi o cheie numerică, fie ea $y = 7$. Conform acestei instanţe algoritmul va furniza ca rezultat faptul că elementul $x[5]$ coincide cu cheia dată (soluţie a problemei).

Un algoritm este ***corect*** dacă pentru orice instanţă a sa el se încheie cu o ieşire corectă care este soluţie a problemei de calcul rezolvată prin algoritmul respectiv. Unii algoritmi, deşi sunt incorecţi în sensul că nu conduc la nici o soluţie în timp finit sau conduc la soluţii eronate, ei pot fi utili atât timp cât erorile produse de ei pot fi controlate.

Comportarea aceluiaşi algoritm poate fi diferită în funcţie de datele de intrare. De aceea se impune o mare atenţie asupra acestora din urmă. O variabilă prezentă în intrarea unui algoritm poate identifica un tablou sau un obiect al unei date compuse şi va fi tratată ca un pointer la elementele tabloului, respectiv la câmpurile (atributele) obiectului corespunzător.

Modul în care se defineşte ***dimensiunea datelor de intrare*** depinde de problema de calcul analizată. Ea poate fi exprimată prin:

a) numărul de elemente cuprinse în datele de intrare (de exemplu, dimensiunea unui tablou de numere întregi);

b) numărul total de biţi din reprezentarea binară a datelor de intrare (vezi algoritmul de înmulţire a două numere mari întregi);

c) două numere naturale (de exemplu, pentru reprezentarea unui graf se solicită numărul de vârfuri şi numărul de muchii ale sale);

d) valoarea numerică a intrării (pentru algoritmi numerici, de exemplu algoritmul „La Russe").

Pentru fiecare algoritm care rezolvă o anumită problemă P este necesar a se preciza modul de exprimare a dimensiunii datelor de intrare.

Dacă există un algoritm care rezolvă problema P nu înseamnă că el este unic.

De exemplu, există algoritmi ca: QuickSort (sortare rapidă), MergeSort (sortare prin fuziune), TreeSort (sortare prin arbori), sortare prin selecţie şi inserţie etc. care sunt utilizaţi în acelaşi scop.

Prin urmare, apare necesitatea alegerii unui algoritm din clasa de algoritmi care rezolvă problema P care să corespundă unor cerinţe. Algoritmul depinde de aplicaţie, implementare, mediu, frecvenţa utilizării etc. Compararea algoritmilor este un proces subtil care are în vedere mai multe aspecte. În continuare, vom căuta să analizăm câteva din aceste aspecte care joacă un rol important în elaborarea unui algoritm.

2. COMPLEXITATEA CALCULULUI

Teoria complexităţii constituie un domeniu foarte important al teoriei algoritmilor care investighează rezolvabilitatea individuală a problemelor sub anumite restricţii de resurse.

Teoria complexităţii calculului cuprinde:

1) Un studiu analitic al complexităţii unor clase de probleme în raport cu diverse tipuri concrete de maşini matematice care le rezolvă;

2) Un studiu calitativ al complexităţii algoritmilor care rezolvă astfel de probleme.

Mai precis, ***complexitatea calculului*** se bazează pe legăturile existente între ***problemele rezolvabile algoritmic*** şi ***resursele de calcul***.

Înainte de a trece la analiza unui algoritm, trebuie să se cunoască o tehnologie de implementare a acestuia care va include un model pentru resursele sale şi costurile corespunzătoare. Aşadar, ***resursele de calcul*** sunt introduse prin definirea unui model de calcul care, de regulă, reprezintă o maşină. În acest sens pot fi utilizate: ***maşini Turing de diverse tipuri, maşini cu acces aleatoriu (random-acces machine, RAM), circuite booleene*** etc. Resursele tipice sunt: ***timp, spaţiu, benzi, capete de citire/scriere, nivele într-un circuit*** etc.

Pentru tratarea celor două elemente fundamentale, ***problemă şi maşină,*** considerăm trei nivele abstracte în complexitatea calculului.

a) Un nivel înalt care pune în evidenţă *complexitatea abstractă* a calculului, cunoscută sub numele de ***„teoria Blum de complexitate a calculului"***, care este independentă atât de problemă cât şi de maşină. O astfel de teorie este construită pe o enumerare efectivă a tuturor funcţiilor parţial recursive, o enumerare a aşa numitelor funcţii „ cost-măsură " şi două axiome de bază (axiomele lui Blum).

b) Un nivel mediu care pune în evidenţă *complexitatea structurală,* independentă de problemă dar dependentă de maşină. Scopul unei astfel de complexităţi este de a studia capabilitatea unor modele de calcul concrete pentru resurse sau combinaţii de resurse, independent de problema propusă.

c) Un nivel inferior ce indică o *complexitate concretă* care depinde atât de maşină cât şi de problemă. Scopul său este de a detecta o bună evaluare, cu un cost pe cât posibil minim, folosind o resursă (sau combinaţii de resurse) într-un model precizat pentru rezolvarea problemei.

În general, dificultatea unei probleme P se poate măsura prin resursele de calcul limitate asociate algoritmului secvenţial care o rezolvă. Aceste resurse sunt ***timpul şi spaţiul de memorie*** necesare pentru execuţia algoritmului respectiv. O astfel de măsură se poate evalua în diverse moduri. În termenii maşinii Turing, ca model de calcul, complexitatea temporală este stabilită de numărul de deplasări necesare realizării calculului respectiv, iar pentru un computer digital, de numărul de ciclări ale maşinii sau timpul real solicitat de maşină pentru acel calcul. În ceea ce priveşte complexitatea spaţială, pentru maşina Turing, ea constă în numărul de celule (pătrate) ale benzii folosite în calcul, iar pentru computer, în numărul de bytes utilizaţi.

Prin urmare, ideea de ***complexitate a unui algoritm*** poate fi privită sub două aspecte:

a) static, prin structura sa.

b) dinamic, prin durata execuţiei sale.

Complexitatea spaţială se exprimă prin dimensiunea spaţiului de memorie necesar algoritmului, independent de comportarea pe care o are el pe parcursul execuţiei.

Complexitatea temporală se exprimă prin timpul necesar execuţiei sale. Axiomele lui Blum, de exemplu, pun în evidenţă măsuri de complexitate dinamică cu consecinţe ca: teorema corelării măsurilor, teorema accelerării, teorema lacunei etc.

Există numeroase probleme de optimizare în reţele de transport, informatică, electronică, logistica construcţiilor, teoria comunicaţiilor (în codurile corectoare de erori), criptografie etc. Pentru algoritmii de rezolvare a acestor probleme este obligatoriu un studiu al complexităţii lor. Complexitatea algoritmilor relativ la coduri, reprezentarea compactă a mesajelor în vederea transmiterii lor prin diverse canale, permite o analiză asupra eficacităţii metodelor de codificare/decodificare. Relativ la criptografie, studiul complexităţii implică elaborarea unor tehnici care să permită cifrarea mesajelor sub o formă care să le asigure, pe cât posibil, inviolabilitatea.

Analiza unui algoritm conduce la stabilirea eficienţei sale şi implică, în principal, ideea de complexitate pentru că ea prevede resursele solicitate de algoritmul respectiv. Stabilirea complexităţii unui algoritm este necesară, pe de o parte, datorită faptului că mulţi algoritmi sunt eliminaţi de practică deoarece necesită un timp de execuţie şi un spaţiu de memorie foarte mari, iar pe de altă parte, se caută algoritmi cât mai performanţi în vederea rezolvării cu calculatorul a unor probleme destul de complexe impuse de practică. Un calculator, oricât de performant ar fi, este în esenţă un automat finit, deci el are posibilităţi limitate. Performanţa unui algoritm se stabileşte conform unui criteriu care, de cele mai multe ori, îl constituie timpul de execuţie al algoritmului respectiv.

Având în vedere faptul că pentru seturi de date de intrare diferite acelaşi algoritm foloseşte timpi de execuţie diferiţi este necesar a lua în consideraţie:

• ***timpul în cazul cel mai favorabil*** (durata minimă pentru execuţia algoritmului);

• ***timpul mediu*** (raportul dintre suma timpilor necesari pentru toate seturile de date posibile şi numărul acestor seturi);

• ***timpul în cazul cel mai defavorabil*** (durata maximă pentru execuţia algoritmului).

Acesta din urmă oferă o margine superioară a timpului de execuţie pentru toate intrările de o dimensiune fixă şi reprezintă situaţia cea mai indicată pentru căutarea de informaţii într-o bază de date.

Să considerăm implementarea unui algoritm oarecare care să primească intrări de dimensiuni diferite. La o intrare de dimensiune n vom asocia o măsură $f(n)$ de folosire a resurselor sale care, de regulă, trebuie să se minimizeze. În mod normal, $f(n)$ va reprezenta timpul solicitat de algoritm pentru intrarea de dimensiune n în cazul cel mai defavorabil sau mediu (la alegerea noastră). Există o serie de factori care complică mai mult sau mai puţin calculul exact al lui $f(n)$.

De exemplu:

a) Timpul solicitat pentru execuţia unei instrucţiuni individuale dintr-un program, cu aceeaşi instanţă şi pe aceeaşi maşină, poate varia.

b) Poate exista o mare variaţie de folosire a resurselor pentru intrări de o anume dimensiune constantă n.

c) Cazul cel mai defavorabil poate fi mai rar întâlnit în practică, iar cazul mediu să fie nereprezentativ.

d) Unii algoritmi se pot executa mai bine pe anumite clase de intrări.

În acest sens se impun câteva sugestii, cum ar fi:

- Identificarea operaţiilor abstracte folosite de algoritm. Pentru a obţine o maximizare a independenţei de maşină este necesară o analiză a acestor operaţii abstracte, pentru că resursele solicitate de o mare parte din ele vor fi neimportante. Unele din ele sunt utilizate o singură dată la iniţializare.

- Un algoritm, în general, posedă cel puţin un ciclu. Acesta trebuie identificat pentru că instrucţiunile din interiorul său se execută mult mai des şi ele vor juca un rol important în analiza timpului de execuţie a algoritmului. Prin contorizarea acestor instrucţiuni se determină o margine superioară corespunzătoare pentru valoarea timpului de execuţie în cazul cel mai defavorabil şi, dacă este posibil, o configurare a cazului mediu. Această limită superioară este necesară pentru că, în mod practic, nu poate fi găsită o valoare exactă a timpului de execuţie în cazul cel mai defavorabil.

- Timpul necesar unui algoritm pentru rezolvarea unei probleme cu o instanţă dată, reprezintă numărul operaţiilor elementare (primitive) efectuate de algoritm pentru acea instanţă. Se acceptă o astfel de interpretare deoarece se presupune că execuţia unei astfel de operaţii se produce într-un timp constant şi nu depinde de mărimea operanzilor şi nici de timpul de memorare a rezultatului. De aceea resursa timp asociată se analizează independent de sistemul de calcul pe care se implementează algoritmul respectiv.

- Deşi progresele tehnologice actuale fac ca necesarul de memorie să treacă pe un plan secund, există totuşi situaţii în care spaţiul de memorie utilizat este folosit ca argument în stabilirea unor limite inferioare pentru timpul de execuţie.

- Viteza de calcul ce caracterizează actuala generaţie de calculatoare are drept consecinţă faptul că problema timpului de execuţie se pune, în general, pentru valori foarte mari ale dimensiunii intrării. Aceasta conduce la ideea unei analize a termenului dominant (cel care tinde cel mai repede la infinit) din formula de exprimare a numărului de operaţii elementare executate de către algoritm.

Dimensiunea informaţiilor prelucrate în majoritatea problemelor pe care le întâlnim în practică nu rămâne constantă.

Pentru a înţelege noţiunea de ”procedeu mecanic de calcul“ au fost propuse numeroase modele de calcul formal. Conform celebrei teze a lui Church - tot ceea ce este intuitiv calculabil cu ajutorul unui algoritm este calculabil indiferent de modelul formal de calcul - aceste modele sunt echivalente. În mod clasic, în teoria complexităţii, se consideră drept model maşina Turing care prezintă dublul avantaj de a constitui un model de program pentru calculator, dar şi un cronometru pentru timpul de execuţie al său. Conform acestui model, complexitatea unui algoritm **A**, pentru orice dimensiune n a intrării, este o funcţie $f(n)$ ce exprimă timpul maxim de execuţie a algoritmului.

În descrierea algoritmilor, cel mai convenabil model este modelul RAM având în vedere şi cele mai importante clase de complexitate studiate (timp şi spaţiu polinomial).

Există câteva funcţii tipice de exprimare a timpului de execuţie a unui algoritm. Putem obţine, în mod obişnuit, o astfel de funcţie printr-o relaţie de recurenţă.

Cel mai adesea se va întâlni funcţia de forma

$f(n) = cg(n) + tm$ unde $c > 0$ este o constantă, n reprezintă dimensiunea intrării, iar tm este un “termen mic” care este semnificativ doar pentru valori mici ale lui n sau pentru algoritmi sofisticaţi.

Funcţia $g(n)$ poate fi:

• constantă (algoritmul se execută în timp constant);
• log n (algoritmul se execută în timp log);
• n^m (m = 0, 1, 2…)(algoritmul se execută în timp polinomial);

Cazuri particulare:

a) n (algoritmul se execută în timp liniar);
b) nlog n (algoritmul se execută în timp liniar log);
c) n^2 (algoritmul se execută în timp pătratic);
d) n^3 (algoritmul se execută în timp cubic);
• $n!$ (algoritmul se execută în timp factorial);
• k^n ($k > 1$, constantă) (algoritmul se execută în timp exponenţial).

Exemplul 2.1.

Să considerăm un algoritm care efectuează operaţii de punere/extragere a unei date de 32 biţi efectuate pe o stivă cu elemente întregi. Astfel de operaţii vor necesita un timp constant. Un algoritm care se execută într-un timp constant mic se zice că este ***“perfect”.***

Exemplul 2.2.

Să presupunem un algoritm a cărui intrare constă dintr-un şir de n caractere. El poate prelucra intrarea sa caracter cu caracter, solicitând acelaşi timp pentru fiecare astfel de caracter. În acest caz, $f(n) = n$. Un astfel de algoritm care se execută în timp liniar se zice că este ***“excelent”.***

Exemplul 2.3.

Un algoritm poate cicla direct intrarea sa formată din numere întregi sub forma

$$f(n) = n + f(n - 1).$$

Iterând această relaţie se obţine $f(n) = n + f(n - 1) = n + ((n-1) + f(n - 2)) =.....$
$= n + (n - 1) + (n - 2) + ... + 2 + f(1) = n^2/2 + n/2 + k$, unde k este o constantă. Pentru un n suficient de mare, k şi $n/2$ sunt mici faţă de n^2. Un astfel de algoritm se execută într-un timp pătratic.

Un algoritm care solicită o prelucrare a celor n^2 perechi de caractere ce corespund unei intrări care constă dintr-un şir de n caractere, solicită tot un timp pătratic de execuţie.

Pentru intrări de dimensiuni mari algoritmii care se execută în timp pătratic vor evolua mai lent.

Exemplul 2.4.

Un algoritm poate prelucra, la fiecare pas, jumătate din intrarea sa. Aşadar,

$$f(n) = f(n/2) + 1$$

(cu aproximare în cazul în care n nu este o putere naturală a lui 2).

Fie $n = 2^x$. Atunci, $f(n) = f(2^x) = 1 + f(2^{x-1}) = 1 + (1 + f(2^{x-2})) = ... = x + f(2^0) = x + k$, unde k este o constantă.

În acest caz, $f(2^x)$ este aproximativ x, iar $f(n)$ este aproximativ $\log_2 n$. Un astfel de algoritm se execută în timp ***log.*** Aceşti algoritmi au o creştere foarte lentă şi sunt foarte buni în practică.

Este foarte cunoscut algoritmul "Divide et Impera" folosit în Quicksort, Mergesort etc. Pentru acest algoritm, $f(n) = n + 2f(n/2)$.

Pentru $n = 2^x$, folosind un mic artificiu, se obţine:

$f(2^x)/2^x = 2^x/2^x + 2*f(2^x/2)/2^x = 1 + f(2^{x-1})/2^{x-1} = 1 + (1 + f(2^{x-2})/2^{x-2}) = ...= x + f(2^0)/2^0 =$
$= x+k$, unde k este o constantă.

Rezultă că $f(2^x) = 2^x(x + k) = 2^x x + tm$. Prin urmare, $f(n) = n\log_2 n + tm$. În concluzie, algoritmul respectiv se execută în timp ***liniar log.***

Exemplul 2.5.

Să presupunem că intrarea unui algoritm constă într-o mulţime de n numere întregi. Se cere să se găsească o submulţime a acestei mulţimi cu proprietatea că suma elementelor sale este nulă.

Printr-o cercetare exhaustivă, în cazul cel mai defavorabil, se vor verifica toate cele 2^n submulţimi posibile. Un astfel de algoritm se va executa într-un timp exponenţial.

Exemplul 2.6.

Un algoritm, prin care se cere obţinerea tuturor anagramelor corespunzătoare unui cuvânt format din n caractere la intrare, se va executa într-un timp factorial, pentru că există $n!$ astfel de posibilităţi.

Funcţia $n!$ creşte aproximativ cu aceeaşi viteză ca n^n, dar mai repede decât 2^n.

În 1964, Cobham a introdus clasa P a ***problemelor rezolvabile algoritmic în timp polinomial***, adică o problemă ce se rezolvă printr-un algoritm **A** care pentru orice număr natural n, funcţia sa de complexitate este mărginită superior de un polinom cunoscut $p(n)$ în variabila n, adică $f(n) \leq kp(n)$, unde k este o constantă nenulă. În această clasă se întâlnesc aşa numitele ***probleme "facile***". Se mai spune despre un algoritm de complexitate polinomială că reprezintă rezultatul unei cunoaşteri detaliate a problemei pe care o rezolvă.

Despre un algoritm care nu este de complexitate polinomială se zice că se comportă "***exponenţial".*** În aceeaşi clasă a algoritmilor exponenţiali sunt încadraţi şi algoritmii de complexitate $n\log n$ deşi nu corespund în sensul strict matematic al noţiunii respective. Având în vedere viteza de creştere sau ordinul de creştere a timpului de execuţie în raport cu n a funcţiilor exponenţiale faţă de cele polinomiale, în general, algoritmii exponenţiali sunt inutilizabili în practică. Despre un algoritm de complexitate exponenţială se spune că este o variantă a unei enumerări totale a căilor de identificare a soluţiilor problemei respective.

Există unii algoritmi cu comportare exponenţială care, pentru valori relativ mici ale lui n, sunt mai eficienţi decât cei alternativi cu comportare polinomială. De asemenea, unii algoritmi exponenţiali se comportă acceptabil pentru unele probleme particulare (vezi metoda simplex de rezolvare a problemelor de programare liniară).

În general, despre o problemă pentru care s-a dovedit că nu există un algoritm de complexitate polinomială pentru rezolvarea ei, adică nu este ***tractabilă,*** se spune că este ***intractabilă.*** Dificultatea unei astfel de probleme trebuie remarcată prin faptul că

timpul necesar pentru găsirea unei soluţii este de ordin exponenţial, adică funcţia nu se poate exprima printr-o expresie care să poată fi mărginită superior de un polinom în variabila n.

Dacă se are în vedere ca model, maşina de calcul în paralel (capabilă de a efectua simultan un număr relativ mare de calcule independente), ***clasa problemelor dificile*** (cele care nu sunt facile) se poate împărţi în două subclase:

a) subclasa problemelor pentru care nu există algoritmi nedeterminişti de complexitate polinomială pentru rezolvarea lor;

b) subclasa problemelor pentru care există algoritmi nedeterminişti de complexitate polinomială pentru rezolvarea lor (***NP***).

Trebuie observat faptul că majoritatea problemelor care par a fi dificile admit algoritmi nedeterminişti de complexitate polinomială pentru rezolvarea lor. Noţiunea de ***"algoritm determinist"*** trebuie înţeleasă în sensul că structura sa nu permite o alegere între mai multe căi posibile în vederea obţinerii rezultatului dorit. Conceptul de ***"algoritm nedeterminist"*** trebuie înţeles în sensul că el se poate afla în mai multe stări independente care nu se pot alege după anumite criterii şi nici genera aleator. Un algoritm se numeşte ***nedeterminist polinomial*** dacă complexitatea calculelor, efectuate pe orice cale a arborelui ce descrie ramificările procesului de căutare a soluţiei, este polinomială.

Având în vedere aceste interpretări, conceptul de ***algoritm determinist polinomial*** este evident.

Clasa problemelor ***"dificile în sensul cel mai tare"*** este formată din acele probleme pentru care s-a demonstrat că nu există algoritmi pentru rezolvarea lor (de exemplu, problema a zecea a lui Hilbert cu privire la rezolvabilitatea în numere întregi a ecuaţiilor polinomiale).

3. COMPLEXITATE ASIMPTOTICĂ

O simplă caracterizare a eficienţei unui algoritm constă în viteza de creştere a timpului său de execuţie care oferă posibilitatea de a compara performanţele relative ale unor algoritmi alternativi.

În general, este destul de dificil de a determina expresia exactă ce defineşte timpul de execuţie al unui algoritm în funcţie de dimensiunea problemei, de aceea se stabilesc anumite limite între care el poate varia. Complexitatea se exprimă în funcţie de această dimensiune.

Când datele de intrare au dimensiuni suficient de mari, astfel încât numai viteza de creştere a timpului de execuţie să fie relevantă, vom studia eficienţa ***asimptotică*** a algoritmilor. Mai exact, se va urmări viteza de execuţie a mai multor algoritmi cu acelaşi obiectiv, sau, altfel zis, viteza de creştere a funcţiilor a căror expresie reprezintă timpul de execuţie a unor astfel de algoritmi. În practică există un interes deosebit asupra modului de creştere a timpului de execuţie a unui algoritm o dată cu creşterea dimensiunii datelor de intrare până la cazul limită în care această dimensiune creşte la infinit. Se foloseşte, în acest sens, conceptul de ***timp asimptotic de execuţie a unui algoritm.*** Pentru exprimarea unui astfel de timp se va folosi un limbaj riguros care cuprinde următoarele simboluri asimptotice: Θ, O , o , Ω, ω. Se vor folosi funcţii al căror domeniu de definiţie îl reprezintă mulţimea numerelor naturale N. Notaţia se poate extinde, în unele situaţii, la domeniul numerelor reale sau restrânge la o submulţime a mulţimii numerelor naturale.

Fie $\Re$ mulţimea funcţiilor nenegative care au ca domeniul de definiţie mulţimea numerelor naturale N = {0, 1, 2, ...}.

Fie funcţiile $f(n)$, $g(n)$ $f(n) \in \Re,\ g(n) \in \Re$.

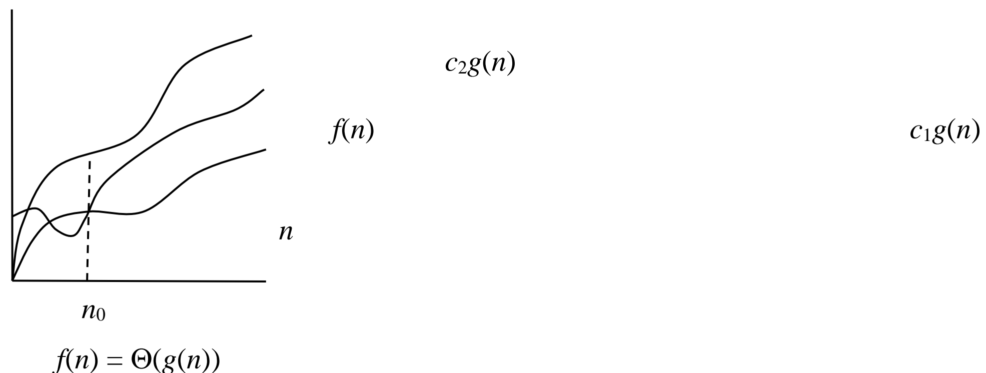
3.1. Θ - notaţia

Definiţia 3.1.1. $f(n) = \Theta(g(n))$ $(n \to \infty)$ dacă există constantele $c1 > 0$, $c2 > 0$ şi $n_0 \in N$, astfel încât $\forall n \geq n_0,\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$.

Notaţia $n \to \infty$ indică, o dată în plus, faptul că relaţia asimptotică are loc pentru valori suficient de mari ale lui n. Într-o scriere asimptotică, chiar dacă nu este prezentă această notaţie cu scopul de a simplifica scrierea, ea este subînţeleasă.

Din punct de vedere matematic este corect a folosi notaţia $f(n) \in \Theta(g(n))$, pentru că $\Theta(g(n)) = \{f(n)$/există constantele $c_1 > 0$, $c_2 > 0$ şi $n_0 \in N$

, astfel încât $\forall n \geq n_0,\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$.



$f(n) = \Theta(g(n))$

În literatura de specialitate, se foloseşte, printr-un abuz de utilizare a semnului de egalitate, notaţia $f(n) = \Theta(g(n))$ $(n \to \infty)$. Se acceptă ambele notaţii şi pentru celelalte simboluri care vor urma. Se spune despre funcţiile $f(n)$ şi $g(n)$ că au aproape aceeaşi viteză de creştere având în vedere constantele pozitive multiplicative c_1, c_2 sau că ele sunt funcţii egale până la un factor constant. Vom spune că funcţia $g(n)$ constituie o ***margine asimptotică tare sau strânsă*** (inferioară şi superioară) pentru funcţia $f(n)$.

Din definiţia mulţimii $\Theta(g(n))$ se observă că este necesar ca elementele sale să fie funcţii nenegative pentru valori ale lui n suficient de mari. O funcţie asimptotic pozitivă este strict pozitivă pentru orice valoare a lui n suficient de mare $(n \geq n_0)$.

În mod intuitiv, pe baza definiţiei simbolului Θ, într-o expresie polinomială ce reprezintă timpul de execuţie a unui algoritm se ia în calcul doar termenul dominant (de ordin maxim) şi se neglijează toţi termenii de ordin inferior cât şi coeficientul termenului dominant.

Fie $f(n)= (3/7)n^2- 4n$. Pentru a arăta că $f(n)= \Theta(n^2)$ trebuie să determinăm constantele pozitive n_0, c_1, c_2 astfel încât

$0 \leq c_1 n^2 \leq (3/7)n^2 - 4n \leq c_2 n^2$. Pentru $\forall n \geq 1,\ c_2 \geq 3/7$ se observă că are loc inegalitatea $3/7 - 4/n \leq c_2$, iar pentru $\forall n \geq 1,\ c_1 \leq 1/35$ are loc inegalitatea $3/7 - 4/n \geq c_1$. În concluzie, există constantele

$c_1 = 1/35$, $c_2 = 3/7$ şi $n_0 = 10$ astfel încât să aibă loc dubla inegalitate, adică $f(n) = \Theta(n^2)$. Adesea, dacă i se dă constantei c_1 o valoare ceva mai mică decât coeficientul termenului dominant, iar lui c_2 o valoare puţin mai mare decât acelaşi coeficient, definiţia simbolului Θ ar putea fi îndeplinită. Coeficientul termenului dominant poate fi ignorat pentru că el ar modifica cele două constante c_1 şi c_2 doar cu un factor constant.

Să observăm că aceste constante, pentru o aceeaşi funcţie, nu sunt unice şi ele depind de funcţia respectivă.

Prin relaţia $0 \le f(n) \le cg(n)$, se înţelege că funcţia $f(n)$ are ordinul de creştere al funcţiei $g(n)$ pentru valori ale lui n suficient de mari, până la un anumit factor constant c. Logaritmând această relaţie, se obţine $log(f(n)) \le c_1 + log(g(n))$ (c_1 = log c, este o constantă). Logaritmând şi relaţia $0 \le c'g(n) \le f(n)$, se obţine $c_2 + log(g(n)) \le log(f(n))$ (c_2= logc' este o constantă). Se observă că $f(n) = \Theta(g(n))$ ($n \to \infty$) dacă şi numai dacă există o constantă c astfel încât pentru un n suficient de mare, log($f(n)$) şi log($g(n)$) să difere prin cel mult o astfel de constantă multiplicativă.

Se spune că implementarea unui algoritm se execută în timp ***log,*** respectiv în timp ***liniar,*** dacă funcţia sa de complexitate $f(n)$ este din $\Theta(\log(n))$, respectiv $\Theta(n)$.

3.2. O - notaţia

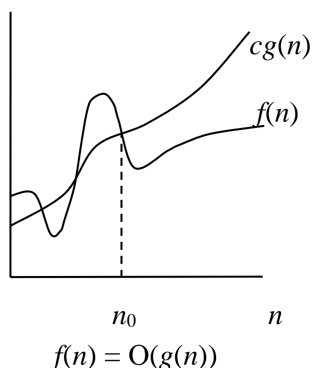
Definiţia 3.2.1 $f(n) = O(g(n))$ ($n \to \infty$) dacă $\exists\, c > 0$, $\exists\, n_0 \in N$ astfel încât $\forall n \ge n_0$, $0 \le f(n) \le c\ \ g(n)$.

Printr-o astfel de notaţie se pune în evidenţă faptul că funcţia $f(n)$ creşte strict mai lent sau cel mult la fel ca funcţia $g(n)$.

Se mai spune că funcţia $g(n)$ domină funcţia $f(n)$ sau că funcţia $f(n)$ este dominată de funcţia $g(n)$). Cu alte cuvinte, prin notaţia $f(n) = O(g(n))$ ($n \to \infty$) înţelegem faptul că un multiplu constant al lui $g(n)$ constituie o margine asimptotică superioară a lui $f(n)$. Aşadar

$$O(g(n)) = \{f(n) \in \Re / \exists c > 0,\ \exists n_0 \in N$$

astfel încât $0 \le f(n) \le g(n),\ \forall n \ge n_0\}$.



$f(n) = O(g(n))$

În literatura de specialitate există o distincţie clară între o margine asimptotică superioară şi o margine asimptotică tare (strânsă) pe care o oferă Θ. Notaţia O oferă o margine asimptotică superioară care nu este neapărat o margine asimptotică tare.

Exemplul 3.2.1.

$\mathbf{5n^2 = O(n^2)}$ constituie o margine asimptotică superioară tare.

$\mathbf{5n = O(n^2)}$ este o margine asimptotică superioară.

Din faptul că $f(n) = \Theta(g(n))$ rezultă că $f(n) = O(g(n))$ pentru că notaţia Θ este mai puternică decât notaţia O, adică $\Theta(g(n)) \subseteq O(g(n))$.

O funcţie liniară $f(n) = an + b = \Theta(n)$ ($a > 0$). În schimb, folosind notaţia O, este adevărată şi relaţia $f(n) = O(n^2)$. Într-adevăr trebuie să determinăm cele două constante strict pozitive n_0 şi c astfel încât să aibă loc relaţia $0 \le an + b \le cn^2$ pentru orice $n \ge n_0$. Dacă dreapta intersectează parabola trebuie ca ecuaţia ataşată să aibă rădăcini reale. De aceea, vom considera expresia $a^2 + 4bc$ de forma $(a + 2b)^2$. Printr-un calcul elementar se observă că pentru orice $n \ge n_0 = 1$ şi $c = a + |b|$ $(c > 0)$ relaţia este adevărată. Dacă dreapta nu intersectează parabola, relaţia cerută este evident adevărată.

Printr-o afirmaţie de genul „***timpul de execuţie al unui algoritm A este O(n^k)***" se înţelege că timpul de execuţie în cazul cel mai defavorabil (exprimat printr-o funcţie de n) este $O(n^k)$, adică oricare ar fi datele de intrare de dimensiune n, pentru fiecare valoare a lui n, timpul de execuţie respectiv este $O(n^k)$, unde k este o constantă întreagă.

Propoziţia 3.2.1. O condiţie suficientă ca o funcţie $g(n)$ să dea o comportare asimptotică de tip O pentru o funcţie $f(n)$, adică $f(n) = O(g(n))$, este să existe o constantă $n_0 \in N$ astfel încât pentru orice $n \ge n_0$ să fie îndeplinite următoarele condiţii:

a) $g(n) > 0$;

b) $lim_{n\to\infty}\frac{f(n)}{g(n)}$ să existe şi să fie finită.

Demonstraţie: Fie $l = lim_{n\to\infty}\frac{f(n)}{g(n)}$. Aceasta înseamnă că $\forall\varepsilon > 0, \exists n_0(\varepsilon)$ astfel încât $\left|\frac{f(n)}{g(n)} - l\right| < \varepsilon, \forall n \geq n_0(\varepsilon)$. Adică $-\varepsilon < \frac{f(n)}{g(n)} - l < \varepsilon \Rightarrow l - \varepsilon < \frac{f(n)}{g(n)} < l + \varepsilon$ .Alegând $\varepsilon < 1 \Rightarrow \frac{f(n)}{g(n)} < l + 1,\ \forall n > n_0(\varepsilon)$. Există două constante pozitive $n_0(\varepsilon),\ n_1^0 \geq l + 1$ astfel încât $\forall n > n_0(\varepsilon),\ g(n) > 0,\ 0 \leq f(n) < n_1^0 * g(n)$. Aşadar $f(n) = O(g(n))$.

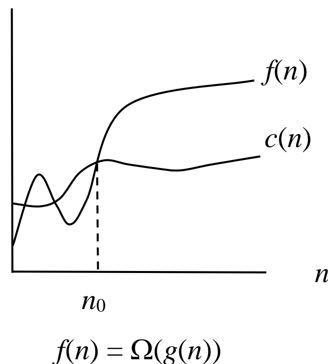
Această condiţie nu este şi necesară. Justificaţi această afirmaţie printr-un contraexemplu.

Exemplul 3.2.2. Fie funcţiile $f(n),\ g(n) \in \Re$ definite prin relaţiile $f(n) = 10n$ şi $g(n) = n^2$. Dacă calculăm $f(n)$ şi $g(n)$ pentru $1 \leq n \leq 9$ se obţine: $f(1) = 10$, $g(1) = 1$; $f(2) = 20$, $g(2) = 4$; $f(3) = 30$, $g(3) = 9$; $f(4) = 40$, $g(4) = 16$,…, $f(9) = 90$, $g(9) = 81$. Pentru $n \geq 10$ se observă că $n^2 \geq 10n$, adică $0 \leq f(n) \leq g(n)$.

3.3. Ω - notaţia

Definiţia 3.3.1 $f(n) = \Omega(g(n))$ $(n\to\infty)$ dacă $g(n) = O(f(n))$ $(n\to\infty)$.

$\Omega(g(n)) = \{f(n) \in \Re | \exists c > 0, \exists n_0 \in N$ astfel încât $f(n) \geq cg(n) \geq 0, \forall n \geq n_0\}$.



$f(n) = \Omega(g(n))$

În studiul algoritmilor, simbolul Ω este util pentru a pune în evidenţă un timp minim posibil de execuţie. Aşadar simbolul Ω este util în precizarea unei margini asimptotice inferioare nu neapărat tare.

Exerciţiu 3.3.1:

Arătaţi că $\Theta(f(n)) = \{g(n) \in \Re | g \in O(f(n)) \cap \Omega(f(n))\}$.

În practică se demonstrează existenţa marginilor asimptotice tari pornind de la margini asimptotice superioare şi inferioare şi nu invers.

3.4. *o* - notaţia

Definiţia 3.4.1 $f(n) = o(g(n))$ $(n\to\infty)$ dacă $\forall c > 0,\ \exists n_0 \in N$ astfel încât $\forall n \geq n_0$, are loc relaţia $0 \leq f(n) < cg(n)$. Prin urmare, $o(g(n)) = \{f(n) \in \Re | \forall c > 0,\ \exists n_0 \in N$ astfel ca $0 \leq f(n) < cg(n),\ \forall n \geq n_0\}$Prin notaţia „***o***" vom semnala o margine asimptotică superioară care nu este o margine asimptotică superioară tare.

Exemplul 3.4.1.

$10n = o(n^2)$, iar $10n^2 \neq o(n^2)$.

Conform definiţiei simbolului ***o*** putem spune că $f(n) = o(g(n))$ $(n\to\infty)$ dacă $lim_{n\to\infty}\frac{f(n)}{g(n)}$ există şi este egală cu 0.

O astfel de notaţie exprimă faptul că funcţia $f(n)$ creşte strict mai lent decât funcţia $g(n)$ pentru valori ale lui n suficient de mari.

Să observăm, de exemplu, că $\sin(n) = o(n)$ implică $\sin(n) = O(n)$, dar $\sin(n) = O(1)$ nu implică $\sin(n) = o(1)$. Ce puteţi afirma în acest sens ?

Propoziţia 3.4.2. O condiţie necesară şi suficientă ca $lim_{n\to\infty} f(n) = 0$ este ca $f(n) = o(1)$

3.5. *ϖ* - notaţia

Definiţia 3.5.1. $f(n) = \varpi(g(n))$ $(n\to\infty)$ dacă $\forall c > 0,\ \exists n_0 \in N$ astfel încât $\forall n \geq n_0$, are loc relaţia $f(n) > cg(n) \geq 0$.

Simbolul ω se raportează la Ω într-un mod asemănător lui o faţă de O.

Aşadar, $\varpi(g(n)) = \{f(n) \in \Re | \forall c > 0,\ \exists n_0 \in N$ astfel ca $f(n) > cg(n) \geq 0,\ \forall n \geq n_0\}$Se observă că $f(n) = \omega(g(n))$ dacă şi numai dacă $g(n) = o(f(n))$.

Aşadar ω desemnează o margine asimptotică inferioară care nu este o margine asimptotică inferioară tare.

Exemplul 3.5.1.
$n^2 = \omega(n)$, iar $n^2 \neq \omega(n^2)$.
Se observă că $f(n) = \omega(g(n))$ implică $lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ (dacă limita există).

3.6. ~ - notaţia

Definiţia 3.6.1.
$f(n) \sim g(n)(n\to\infty)$ dacă există $lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$.

Propoziţia 3.6.1. O condiţie necesară şi suficientă ca $f(n) \sim g(n)$ $(n\to\infty)$ este ca $f(n) = g(n)\,(1+o(1))$.
Demonstraţie:
Dacă $f(n) \sim g(n)$ atunci $lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$ implică

$$lim_{n\to\infty} \frac{(f(n) - g(n))}{g(n)} = lim_{n\to\infty} \left(\frac{f(n)}{g(n)} - 1\right) = lim_{n\to\infty} \frac{f(n)}{g(n)} - 1 = 0$$

Aşadar (f(n)- $g(n)$)/ $g(n)$ =o(1), adică f(n)= $g(n)$ (1+o(1)).
Reciproc se demonstrează asemănător.

Propoziţia 3.6.2. Dacă $f(n)\sim g(n)$ $(n\to\infty)$, atunci $f(n)=O(g(n))(n\to\infty)$.
Demonstraţie: exerciţiu!

4. Compararea asimptotică a funcţiilor

Multe dintre proprietăţile relaţiilor dintre numerele reale se aplică şi la compararea asimptotică a funcţiilor. Pentru cele ce urmează vom presupune că $f(n)$ şi $g(n)$sunt asimptotic pozitive.

Tranzitivitate :
$f(n) = \Theta(g(n))$ şi $g(n) = \Theta(h(n))$ implică $f(n) = \Theta(h(n))$,
$f(n) = O(g(n))$ şi $g(n) = O(h(n))$ implică $f(n) = O(h(n))$,
$f(n) = \Omega(g(n))$ şi $g(n) = \Omega(h(n))$ implică $f(n) = \Omega(h(n))$,
$f(n) = o(g(n))$ şi $g(n) = o(h(n))$ implică $f(n) = o(h(n))$,
$f(n) = \omega(g(n)$ şi $g(n) = \omega(h(n))$ implică $f(n) = \omega(h(n))$.

Reflexivitate:
$f(n) = \Theta(f(n))$,
$f(n) = O(f(n))$,
$f(n) = \Omega(f(n))$,

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

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

Deoarece aceste proprietăţi sunt valide pentru notaţii asimptotice, se poate trasa o analogie între compararea asimptotică a două funcţii f şi g şi compararea asimptotică a două numere reale a şi b:
$f(n) = O(f(n)) \approx a \le b$,
$f(n) = \Omega(f(n)) \approx a \ge b$,
$f(n) = \Theta(f(n)) \approx a = b$,
$f(n) = o(f(n)) \approx a < b$,
$f(n) = \omega(f(n)) \approx a > b$.

O proprietate a numerelor reale, totuşi, nu se transpune la notaţii asimptotice:
Trihotomia: pentru orice două numere reale a şi b exact una dintre următoarele relaţii este adevărată: $a < b$, $a = b$, $a > b$.

Deşi orice două numere reale pot fi comparate, nu toate funcţiile sunt asimptotic comparabile. Cu alte cuvinte, pentru două funcţii $f(n)$ şi $g(n)$, se poate întâmpla să nu aibă loc nici $f(n) = O(g(n))$ **,** nici $f(n) = \Omega(g(n))$ **.** De exemplu, funcţiile n şi $n^{1+\sin n}$ nu pot fi comparate utilizând notaţii asimptotice, deoarece valoarea exponentului în $n^{1+\sin n}$ oscilează între 0 şi 2, luând toate valorile intermediare.

EXERCIŢII:
Exemplul 3.1.1.
$(n + 1)^3 = \Theta(n^3)$ (1)
$(5 + (2n)^{0.5})^{0.5} = \Theta(n^{0.25})$ (2)

$(1+7/n)^n = \Theta(1)$ (3)

Exemplul 3.4.2.

$n^4 = o(n^7)$ (1)

$5\log n = o(n^{0.03})$ (2)

$\sin n = o(n)$ (3)

$5.2\, n^{0.5} = o(n/8 + 9 \sin n)$ (4)

Exemplul 1.3.9.

$n^4 + 3n^3 - 2 \sim n^4$ (1)

$(5n + 1)^2 \sim 25n^2$ (2)

$\frac{(6n^5+7n^2+1)}{(3n^3+5)} \sim 2n^2$ (3)

$4^n + 8\log n + 78 \sim 4^n$ (4)

$\sin 1/n \sim 1/n$ (5)

$n + \sin n \sim n$ (6)

5. RELAŢII RECURENTE

Cel mai important câştig al exprimării recursive este faptul că ea este naturală şi compactă, fără să ascundă esenţa algoritmului prin detaliile de implementare. Pe de altă parte, apelurile recursive trebuie folosite cu discernământ, deoarece solicită şi ele resursele calculatorului (timp şi memorie). Analiza unui algoritm recursiv implică rezolvarea unui sistem de recurente. Vom vedea în continuare cum pot fi rezolvate astfel de recurente.

Când un algoritm conţine o apelare recursivă la el însuşi, timpul său de execuţie poate fi descris adesea printr-o recurenţă. O *recurenţă* este o ecuaţie sau o inegalitate care descrie întregul timp de execuţie al unei probleme de dimensiune n cu ajutorul timpilor de execuţie pentru date de intrare de dimensiuni mici. Putem, apoi, folosi instrumente matematice pentru a rezolva problema de recurenţă şi pentru a obţine margini ale performanţei algoritmului.

O recurenţă pentru timpul de execuţie al unui algoritm de tipul divide şi stăpâneşte se bazează pe cele trei etape definite în descrierea metodei. La fel ca până acum, vom nota cu $T(n)$ timpul de execuţie al unei probleme de dimensiune n. Dacă dimensiunea problemei este suficient de mică, de exemplu $n \leq c$ pentru o anumită constantă c, soluţia directă ia un timp constant de execuţie, pe care îl vom nota cu $\Theta(1)$. Să presupunem că divizăm problema în a subprobleme, fiecare dintre acestea având dimensiunea de $1/b$ din dimensiunea problemei originale. Dacă $D(n)$ este timpul necesar pentru a divide problema în suprobleme, iar $C(n)$ este timpul necesar pentru a combina soluţiile subproblemelor în soluţia problemei originale, obţinem recurenţa

$$T(n) = \begin{cases} \Theta(n), & n \leq c \\ aT\left(\frac{n}{b}\right) + D(n) + C(n), & n > c \end{cases}$$

5.1 Metoda master

Metoda master furnizează o „reţetă” pentru rezolvarea recurenţelor de forma

$$T(n)= aT(n/b)+ f(n) \quad (5.1)$$

unde $a \geq 1$ şi $b>1$sunt constante, iar $f(n)$ este o funcţie asimptotic pozitivă. Metoda master pretinde memorarea a trei cazuri, dar apoi soluţia multor recurenţe se poate determina destul de uşor, de multe ori fără creion şi hârtie.

Recurenţa (1.4.1) descrie timpul de execuţie al unui algoritm care împarte o problemă de dimensiune n în a subprobleme, fiecare de dimensiune b/n, unde a şi b sunt constante pozitive. Cele a subprobleme sunt rezolvate recursiv, fiecare în timp $T(n/b)$. Costul divizării problemei şi al combinării rezultatelor subproblemelor este descris de funcţia $f(n)$ (Adică, utilizând notaţia $f(n)=D(n) + C(n)$). Din punctul de vedere al corectitudinii tehnice, recurenţa nu este, de fapt, bine definită, deoarece n/b ar putea să nu fie întreg. Înlocuirea fiecăruia dintre cei a termeni $T(n/b)$ fie cu $T(\lfloor n/b \rfloor)$ fie cu $T(\lceil n/b \rceil)$ nu afectează, totuşi, comportamentul asimptotic al recurenţei. În mod normal, ne va conveni, de aceea, să omitem funcţiile parte întreagă inferioară şi superioară când scriem recurenţe divide şi stăpâneşte de această formă.

Teorema master

Metoda master depinde de următoarea teoremă.

Teorema 5.1.1. (*teorema master*) Fie $a \geq 1$ şi $b>1$ constante, fie $f(n)$ o funcţie şi fie $T(n)$ definită pe întregii nenegativi prin recurenţă

$$T(n)= aT(n/b) + f(n)$$

unde interpretăm n/b fie ca $\lfloor n/b \rfloor$ fie ca $\lceil n/b \rceil$. Atunci $T(n)$ poate fi delimitată asimptotic după cum urmează.

1. Dacă $f(n) = O(n^{\log_b^{a-\varepsilon}})$ pentru o anumită constantă $\varepsilon >0$, atunci $T(n) = \Theta(n^{\log_b^a})$.
2. Dacă $f(n) = \Theta(n^{\log_b^a})$, atunci $T(n) = \Theta(n^{\log_b^a} \lg n)$.
3. Dacă $f(n) = \Omega\left(n^{\log_b^{a+\varepsilon}}\right)$, pentru o anumită constantă $\varepsilon >0$ şi dacă $af(n/b) \le cf(n)$ pentru o anumită constantă $c < 1$şi toţi n suficient de mari, atunci $T(n) = \Theta(f(n))$.

Înainte de a aplica teorema master la câteva exemple, să ne oprim puţin ca să înţelegem ce spune. În fiecare dintre cele trei cazuri, comparăm funcţia $f(n)$ cu funcţia $n^{\log_b a}$. Intuitiv, soluţia recurenţei este determinată de cea mai mare dintre cele două funcţii. Dacă funcţia $n^{\log_b a}$ este mai mare, ca în cazul 1, atunci soluţia este $T(n) = \Theta(n^{\log_b^a})$. Dacă funcţia $f(n)$ este mai mare, ca în cazul 3, atunci soluţia este $T(n) = \Theta(f(n))$. Dacă cele două funcţii sunt de acelaşi ordin de mărime, ca în cazul 2, înmulţim cu un factor logaritmic, iar soluţia este

$$T(n) = \Theta(n^{\log_b^a} \lg n) = \Theta(f(n) \lg n).$$

Dincolo de această intuiţie, există nişte detalii tehnice care trebuie înţelese.

În primul caz, $f(n)$ trebuie nu numai să fie mai mică decât $n^{\log_b a}$, trebuie să fie polinomial mai mică. Adică $f(n)$ trebuie să fie asimptotic mai mică decât $n^{\log_b a}$ cu un factor n^{ε} pentru o anumită constantă $\varepsilon >0$. În al treilea caz, $f(n)$ trebuie nu numai să fie mai mare decât $n^{\log_b a}$, trebuie să fie polinomial mai mare şi, în plus, să verifice condiţia de „regularitate” $af(n/b) \le cf(n)$. Această condiţie este satisfăcută de majoritatea funcţiilor polinomial mărginite pe care le vom întâlni.

Este important de realizat că cele trei cazuri nu acoperă toate posibilităţile pentru $f(n)$.

Există un gol între cazurile 1 şi 2 când $f(n)$ este mai mic decât $n^{\log_b a}$ dar nu polinomial mai mic. Analog există un gol între cazurile 2 şi 3 când $f(n)$ este mai mare decât $n^{\log_b a}$ dar nu polinomial mai mare. Dacă funcţia $f(n)$ cade într-unul dintre aceste goluri, sau când condiţia de regularitate din cazul 3 nu este verificată, metoda master nu poate fi utilizată pentru a rezolva recurenţa.

Exemplu 5.1.1.

$T(n) = 9T(n/3) + n.$

Pentru această recurenţă, avem $a = 9$, $b = 3$, $f(n) = n$ şi astfel $n^{\log_b a}= T(n)= n^{\log_3 9}=\Theta(n^2)$.

Deoarece $f(n)= O(n^{\log_3 9-\varepsilon})$, unde $\varepsilon = 1$, putem să aplicăm cazul 1 al teoremei master şi să considerăm că soluţia este $T(n)= \Theta(n^2)$.

Exemplu 5.1.2.

$T(n)=T(2n/3) + 1.$

Pentru această recurenţă, avem $a=1$, $b= 3/2$, $f(n)=1$ şi $n^{\log_b a}= n^{\log_{3/2} 1}= n^0 =1$. Cazul 2 este cel care se aplică deoarece $f(n)= \Theta(n^{\log_b a})= \Theta(1)$ şi astfel soluţia recurenţei este

$T(n)= \Theta(\lg n)$.

Exemplu 5.1.3.

$T(n) = 3T(n/4) + n\lg n,$

Pentru această recurenţă, avem $a=3$, $b=4$, $f(n)= n\lg n$ şi $n^{\log_b a}=n^{\log_4 3} =O(n^{0,793})$. Deoarece $f(n)=\Omega(n^{\log_4 3+\varepsilon})$, unde $\varepsilon \approx 0.2$, cazul 3 se aplică dacă putem arăta că pentru $f(n)$ este verificată condiţia de regularitate. Pentru n suficient de mare,

$$af(n/b)=3(n/4)\lg (n/4) \le (3/4)n\lg n = cf(n) \text{ pentru } c = 3/4.$$

În consecinţă, din cazul 3, soluţia recurenţei este $T(n)= \Theta(n\lg n)$.

Exemplu 5.1.4.

Metoda master nu se aplică recurenţei

$$T(n)= 2T(n/2) + n\lg n,$$

chiar dacă are forma potrivită: $a=2$, $b=2$, $f(n)= n\lg n$ şi $n^{\log_b a} = n$. Se pare că ar trebui să se aplice cazul 3, deoarece $f(n)= n\lg n$ este asimptotic mai mare decât $n^{\log_b a} = n$, dar nu polinomial mai mare. Raportul $f(n)/n^{\log_b a} = (n\lg n)/n$ este asimptotic mai mic decât n^{ε} pentru orice constantă pozitivă ε. În consecinţă, recurenţa cade în golul dintre cazurile 2 şi 3.

5.2. Metoda ecuaţiilor caracteristice

5.2.1. Recurenţe liniare omogene

Există tehnici care pot fi folosite aproape automat pentru a rezolva anumite clase de recurenţe. Vom considera recurenţe liniare omogene, de forma

$$a_0t_n + a_1t_{n-1} + \ldots + a_kt_{n-k} = 0 \qquad (5.2.1)$$

unde t_i sunt valorile pe care le căutăm, iar coeficienţii a_i sunt constante.

Vom căuta soluţii de forma

$$t_n = x^n$$

unde x este o constanta (deocamdată necunoscută). Încercăm aceasta soluţie în (5.2.1) şi obţinem

$$a_0x^n + a_1x^{n-1} + \ldots + a_kx^{n-k} = 0.$$

Soluţiile acestei ecuaţii sunt fie soluţia triviala $x = 0$, care nu ne interesează, fie soluţiile ecuaţiei

$$a_0x^k + a_1x^{k-1} + \ldots + a_k = 0$$

care este ecuaţia caracteristica a recurenţei (1.4.2).

Presupunând deocamdată ca cele k rădăcini r_1, r_2, ..., r_k ale acestei ecuaţii caracteristice sunt distincte, orice combinaţie liniară

$$t_n = \sum_{i=1}^{k} c_i r_i^n$$

este o soluţie a recurenţei (5.2.1), unde constantele c_1, c_2, ..., c_k sunt determinate de condiţiile iniţiale. Este remarcabil că (1.4.2) are numai soluţii de aceasta formă.

Exemplu 5.2.1.

Recurenţa care defineşte şirul lui Fibonacci:

$$t_n = t_{n-1} + t_{n-2} \qquad n \geq 2$$

iar $t_0 = 0$, $t_1 = 1$. Putem să rescriem această recurenţa sub forma $t_n - t_{n-1} - t_{n-2} = 0$ care are ecuaţia caracteristica $x^2 - x - 1 = 0$ cu rădăcinile $r_{1,2} = (1\pm\sqrt{5})/2$. Soluţia generala are forma

$$t_n = c_1 r_1^n + c_2 r_2^n$$

Impunând condiţiile iniţiale, obţinem

$$c_1 + c_2 = 0, \qquad n = 0$$

$$r_1c_1 + r_2c_2 = 1, \qquad n = 1$$

de unde determinăm

$$c_{1,2} = \pm 1/\sqrt{5}$$

Deci, $t_n = 1/\sqrt{5}\,(r_1^n - r_2^n)$. Observam că $r_1 = \phi = (1+\sqrt{5})/2$, $r_2 = -\phi^{-1}$ şi obţinem $t_n = 1/\sqrt{5}(\phi^n - (-\phi)^{-n})$

Putem să arătăm acum că timpul pentru algoritmul care determină şirul Fibonacci este în $\Theta(\phi^n)$.

Se poate arăta că, daca r este o rădăcina de multiplicitate m a ecuaţiei caracteristice, atunci

$$t_n = r^n,\ t_n = nr^n,\ t_n = n^2r^n,\ \ldots,\ t_n = n^{m-1}r^n$$

sunt soluţii pentru (1.4.2). Soluţia generală pentru o astfel de recurenţă este atunci o combinaţie liniară a acestor termeni şi a termenilor proveniţi de la celelalte rădăcini ale ecuaţiei caracteristice. Din nou, sunt de determinat exact k constante din condiţiile iniţiale.

Exemplu 5.2.2.

Fie recurenţa $t_n = 5t_{n-1} - 8t_{n-2} + 4t_{n-3}$ iar $t_0 = 0$, $t_1 = 1$, $t_2 = 2$. Ecuaţia caracteristica

$$x^3 - 5x^2 + 8x - 4 = 0$$

are rădăcinile 1 (de multiplicitate 1) şi 2 (de multiplicitate 2). Soluţia generala este:

$$t_n = c_1 1^n + c_2 2^n + c_3 n 2^n$$

Din condiţiile iniţiale, obţinem $c_1 = -2$, $c_2 = 2$, $c_3 = -1/2$.

5.2.2. Recurenţe liniare neomogene

Consideram acum recurenţe de următoarea formă mai generală

$$a_0t_n + a_1t_{n-1} + \ldots + a_kt_{n-k} = b^n p(n) \qquad (5.2.2)$$

unde b este o constantă, iar $p(n)$ este un polinom în n de grad d. Ideea generala este că, prin manipulări convenabile, să reducem un astfel de caz la o formă omogenă.

Exemplu 5.2.3.

De exemplu, o astfel de recurenţa poate fi:

$$t_n - 2t_{n-1} = 3^n$$

In acest caz, $b = 3$ si $p(n) = 1$, un polinom de grad 0. O simpla manipulare ne permite să reducem acest exemplu la forma (1.4.2). Înmulţim recurenţa cu 3, obţinând $3t_n - 6t_{n-1} = 3^{n+1}$. Înlocuind pe n cu n+1 în recurenţa iniţială, avem $t_{n+1} - 2t_n = 3^{n+1}$. In fine, scădem aceste doua ecuaţii şi obţinem $t_{n+1} - 5t_n + 6t_{n-1} = 0$.

Am obţinut o recurenţa omogena pe care o putem rezolva ca în secţiunea precedentă. Ecuaţia caracteristică este:

$$x^2 - 5x + 6 = 0$$

adică $(x-2)(x-3) = 0$.

Intuitiv, observăm că factorul $(x-2)$ corespunde parţii stângi a recurenţei iniţiale, în timp ce factorul $(x-3)$ a apărut ca rezultat al manipulărilor efectuate, pentru a scăpa de partea dreaptă.

Generalizând acest procedeu, se poate arăta că, pentru a rezolva (1.4.3), este suficient să luăm următoarea ecuaţie caracteristică:

$$(a_0x^k + a_1x^{k-1} + \ldots + a_k)(x-b)^{d+1} = 0$$

Odată ce s-a obţinut această ecuaţie, se procedează ca in cazul omogen.

5.2.3. Schimbarea variabilei

Uneori, printr-o schimbare de variabilă, putem rezolva recurenţe mult mai complicate. În exemplele care urmează, vom nota cu $T(n)$ termenul general al recurenţei şi cu t_k termenul noii recurenţe obţinute printr-o schimbare de variabilă. Presupunem pentru început ca n este o putere a lui 2.

Exemplu 5.2.4.

Fie recurenţa $T(n) = 4T(n/2) + n$, $n > 1$ în care înlocuim pe n cu 2^k, notăm $t_k = T(2^k) = T(n)$ şi obţinem $t_k = 4t_{k-1} + 2^k$. Ecuaţia caracteristică a acestei recurenţe liniare este

$(x-4)(x-2) = 0$ cu rşi deci,

$$t_k = c_1 4^k + c_2 2^k.$$

Înlocuim la loc pe k cu lg n şi obţinem

$$T(n) = c_1 n^2 + c_2 n$$

Rezultă că

$T(n) \in O(n^2 \mid n$ este o putere a lui 2).

EXERCIŢII:

II. PROIECTAREA ALGORITMILOR
PRELIMINARII

1. TEHNICA DIVIDE ET IMPERA.

În această secţiune vom examina o abordare numită *divide* şi *stăpîneşte*. Vom utiliza această abordare pentru a construi un algoritm de sortare. Unul din avantajele algoritmilor de tipul *divide* şi *stăpîneşte* este acela că timpul lor de execuţie este adesea uşor de determinat folosind tehnici date mai sus.

Mulţi algoritmi au o structură recursivă: pentru a rezolva o problemă dată, aceştia sunt apelaţi o dată sau de mai multe ori pentru a rezolva subproblemele apropiate. Aceşti algoritmi folosesc de obicei o abordare de tipul *divide* şi *stăpîneşte*: ei rup problema de rezolvat în mai multe probleme similare problemei iniţiale, dar de dimensiune mai mică, le rezolvă în mod recursiv şi apoi le combină pentru a crea o soluţie a problemei iniţiale.

Paradigma *divide* şi *stăpîneşte* implică trei paşi la fiecare nivel de recursivitate:

- **Divide** problema într-un număr de subprobleme.
- **Stăpîneşte** subproblemele prin rezolvarea acestora în mod recursiv. Dacă dimensiunile acestora sunt suficient de mici, rezolvă subproblemele în mod uzual, nerecursiv.
- **Combină** soluţiile tuturor subproblemelor în soluţia finală pentru problema iniţială.

1.1. Sortarea prin interclasare.

Algoritmul de sortare prin interclasare urmează îndeaproape paradigma *divide* şi *stăpîneşte*. Intuitiv acesta operează astfel:

Divide: Împarte şirul de n elemente care urmează a fi sortat in două subşiruri de câte $n/2$ elemente.

Stăpîpneşte: Sortează recursiv cele două subşiruri utilizînd sortarea prin interclasare.

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

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

Operaţia principală a algoritmului de sortare prin interclasare este interclasarea a două şiruri sortate, în pasul denumit mai sus *combină*. Pentru aceasta vom utiliza o procedură auxiliară **MERGE**(A,p,q,r), unde A este vector, şi p,q,r sunt indici ai vectorului, astfel încît $p \leq q < r$. Procedura presupune că subvectorii A[$p..q$] şi A[q+1..r] sunt sortaţi. Ea interclasează pentru a forma un subvector sortat care înlocuieşte subvectorul curent A[$p..r$].

Deşi vom lăsa pseudocodul pentru această procedură ca exerciţiu, este uşor de imaginat o procedură de tip **MERGE** al cărei timp de execuţie este de ordinul $\Theta(n)$, în care $n = r - p + 1$ este numărul elementelor interclasate. Revenind la exemplul nostru cu cărţile de joc, să presupunem că avem două pachete de cărţi de joc aşezate pe masă cu faţa în sus. Fiecare dintre aceste două pachete de cărţi este sortat, cartea cu valoarea cea mai mică fiind deasupra. Dorim să amestecăm cele două pachete într-un singur pachet sortat, care să rămînă aşezat pe masă cu faţa în jos. Pasul principal este acela de a selecta cartea cu valoarea cea mai mică dintre cele două aflate deasupra pachetelor (fapt care va face ca o nouă carte să fie deasupra pachetului respectiv) şi de a o pune cu faţa în jos pe locul în care se va forma pachetul sortat final. Repetăm acest procedeu pînă cînd unul din pachete nu este iepuizat. În această fază, este suficient să luăm pachetul rămas şi să-l punem pe pachetul deja sortat, întorcînd toate cărţile cu faţa în jos. Din punctul de vedere al timpului de execuţie, fiecare pas de bază durează un timp constant, deoarece comparăm de fiecare dată doar două cărţi. Deoarece avem de făcut cel mult n astfel de operaţii elementare, timpul de execuţie pentru procedura **MERGE** este $\Theta(n)$.

MERGE- SORT (A, p, r)
1. **if** $p< r$
2. **then** $q \leftarrow [(p + r)/2]$
3. **MERGE-SORT** (A, p, q)
4. **MERGE-SORT** (A, q+1, r)
5. **MERGE** (A, p, q, r)

Acum putem utiliza procedura **MERGE** ca subrutină pentru algoritmul de sortare prin interclasare.

Procedura **MERGE- SORT** (A, p, r) sortează elementele din subvectorul A[p,..r]. Dacă $p \geq r$, subvectorul are cel mult un element şi este, prin urmare, deja sortat. Astfel pasul de divizare este prezent aici prin simplul calcul al unui indice q care împarte A[p..r] în doi subvectori A[p..q], conţinînd [n/2] elemente şi A[q+1..r] conţinînd [n/2] elemente. Pentru a sorta întregul şir A = < A[1], A[2],..A[n]>, vom apela procedura **MERGE-SORT** (A, 1, *lungime* [A]), unde din nou , *lungime* [A] = n. Dacă analizăm modul de operare al procedurii, de jos în sus, cînd n este o putere a lui 2, algoritmul constă din interclasarea perechilor de şiruri de lungime 1, pentru a forma şiruri sortate de

lungime 2, interclasarea acestora în şiruri sortate de lungime 4, şi aşa mai departe, pînă cînd două şiruri sortate de lungime $n/2$ sunt interclasate pentru a forma şirul sortat de dimensiune n. Figura 2.1 ilustrează acest proces.

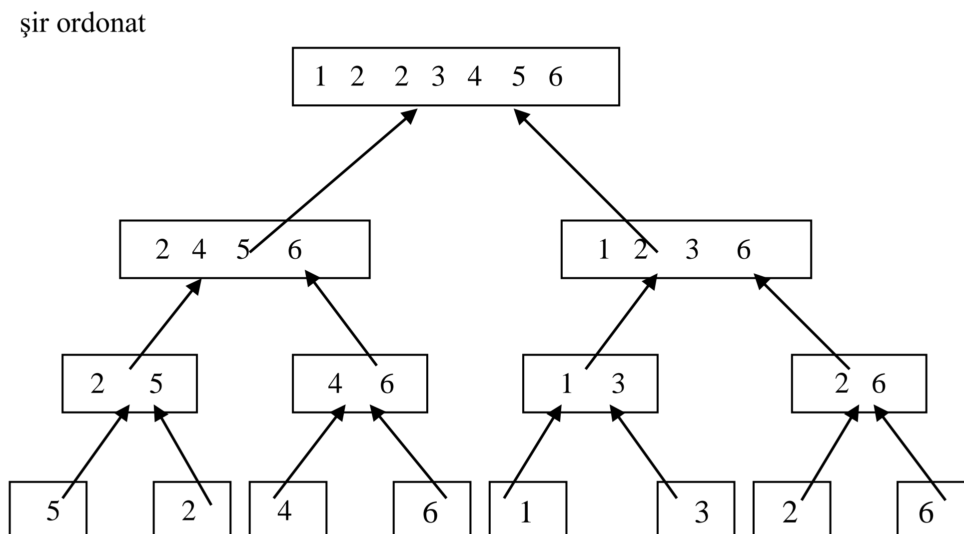


Fig. 1.1. Modul de operare al sortării prin interclasare asupra vectorului A= <5,2,4,6,1,3,2,6>. Lungimile şirurilor sortate, în curs de interclasare, cresc pe măsură ce algoritmul avansează de jos în sus.

1.1.1 Analiza algoritmilor de tipul divide şi stăpîneşte

Cînd un algoritm conţine un apel recursiv la el însuşi, timpul său de execuţie poate fi, adesea, descris printr-o ***relaţie de recurenţă*** sau mai simplu, ***recurenţă***, care descrie întregul timp de execuţie al unei probleme de dimensiune n cu ajutorul timpilor de execuţie pentru date de intrare de dimensiuni mai mici. Putem, apoi, folosi instrumente matematice pentru a rezolva problema de recurenţă şi pentru a obţine margini ale performanţei algoritmului.

O recurenţă pentru timpul de execuţie al unui algoritm de tipul divide şi stăpîneşte se bazează pe cele trei etape definite în descrierea metodei. La fel ca pînă acum, vom nota cu $T(n)$ timpul de execuţie al unei probleme de dimensiune n. Dacă dimensiunea problemei este suficient de mică, de exemplu $n \le c$ pentru o anumită constantă c, soluţia directă ia un timp constant de execuţie, pe care îl vom nota cu $\Theta(1)$. Să presupunem că divizăm problema în a subprobleme, fiecare dintre acestea avînd dimensiunea de b ori mai mică decât dimensiunea problemei originale. Dacă $D(n)$ este timpul necesar pentru a divide problema în suprobleme, iar $C(n)$ este timpul necesar pentru a combina soluţiile subproblemelor în soluţia problemei originale, obţinem recurenţa

$$T(n) = \begin{cases} \Theta(1), & n \le c \\ aT(n/b) + D(n) + C(n), & n > c \end{cases}$$

1.1.2. Analiza sortării prin interclasare

Deşi algoritmul **MERGE-SORT** funcţionează corect şi cînd numărul de elemente nu este par, analiza bazată pe recurenţă se simplifică dacă presupunem că dimensiunea problemei originale este o putere a lui 2. Fiecare pas de împărţire generează deci două subşiruri avînd dimensiunea exact $n/2$. În continuare vom vedea că această presupunere nu afectează ordinul de creştere a recurenţei.

Pentru a determina recurenţa pentru $T(n)$, timpul de execuţie al sortării prin interclasare a n numere, în cazul cel mai defavorabil, vom raţiona în felul următor. Sortarea prin interclasare a unui singur element are nevoie de un timp constant. Cînd avem $n>1$ elemente, vom descompune timpul de execuţie după cum urmează:

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

Stăpîneşte: Rezolvăm recursiv două probleme fiecare de dimensiune $n/2$,care contribuie cu $2T(n/2)$ la timpul de execuţie.

Combină: Am observat, deja că procedura **MERGE** pentru un subvector cu n elemente consumă $\Theta(n)$ timp de execuţie, deci $C(n)= \Theta(n)$.

Cînd adunăm funcţiile $D(n)$ şi $C(n)$ pentru analiza sortării prin interclasare, adunăm o funcţie cu timpul de execuţie $\Theta(1)$. Această sumă este o funcţie lineară în raport cu n, adică are timpul de execuţie $\Theta(n)$. Adăugînd

aceasta la termenul $2T(n/2)$ de la pasul *stăpîneşte*, obţinem timpul de execuţie $T(n)$ in cazul cel mai defavorabil pentru sortarea prin interclasare:

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

Se poate arăta că $T(n)$ este $\Theta(n\lg n)$ unde $\lg n$ reprezintă $\log_2 n$. Pentru numere suficient de mari, sortarea prin interclasare, are timpul de execuţie $\Theta(n\lg n)$.

2. TEHNICA GREEDY

Algoritmii ***greedy*** (greedy = lacom) sunt în general simpli şi sunt folosiţi pentru rezolvarea problemelor de optimizare, cum ar fi: să se găsească cea mai bună ordine de executare a unor lucrări pe calculator, să se găsească cel mai scurt drum într-un graf etc. Algoritmii aplicaţi problemelor de optimizare sunt compuşi dintr-o secvenţă de paşi, la fiecare pas existând mai multe alegeri posibile Un ***algoritm greedy*** va alege la fiecare moment de timp soluţia ce pare a fi cea mai bună la momentul respectiv. Deci este vorba despre o alegere optimă, făcută local, cu speranţa că ea va conduce la un optim global. Acest capitol tratează probleme de optimizare ce pot fi rezolvate cu ajutorul algoritmilor greedy.

Algoritmii greedy conduc în multe cazuri la soluţii optime, dar nu întotdeauna.

2.1. O problemă de selectare a activităţilor

Primul exemplu pe care îl vom considera este o problemă de repartizare a unei resurse mai multor activităţi care concurează pentru a obţine resursa respectivă. Vom vedea ca un algoritm de tip greedy reprezintă o metodă simplă şi elegantă pentru selectarea unei mulţimi maximale de activităţi mutual compatibile.

Să presupunem că dispunem de o mulţime $S = 1,2,..., n$ de n *activităţi* care doresc sa folosească o aceeaşi resursă (de exemplu o sală de lectură). Această resursă poate fi folosita de o singură activitate la un anumit moment de timp. Fiecare activitate i are un *timp de pornire* s_i şi un *timp de oprire* f_i, unde $s_i \le f_i$. Dacă este selecţionată activitatea i, ea se desfăşoară pe durata intervalului $[s_i, f_i)$. Spunem că activităţile i şi j sunt *compatibile* dacă intervalele $[s_i, f_i)$ şi $[s_j, f_j)$ nu se intersectează (adică i şi j sunt compatibile dacă $s_i \ge f_j$ sau $s_j \ge f_i$). *Problema selectării activităţilor* constă din selectarea unei mulţimi maximale de activităţi mutual compatibile.

Un algoritm greedy pentru această problemă este descris de următoarea procedură, prezentată în pseudocod. Vom presupune că activităţile (adică datele de intrare) sunt ordonate crescător după timpul de terminare:

$$f_1 \le f_2 \le ... \le f_n \qquad (2.1)$$

În caz contrar această ordonare poate fi făcută în timp $O(n\lg n)$. Algoritmul de mai jos presupune că datele de intrare s şi f sunt reprezentate ca vectori.

SELECTOR-ACTIVITĂŢI-GREEDY (s, f)
1: $n \leftarrow$ lungime[s]
2: $A \leftarrow \{1\}$
3: $j \leftarrow 1$
4: **for** $i \leftarrow 2$ **to** n **do**
5: **if** $s_i \ge f_j$ **then**
6: $A \leftarrow A \cup \{i\}$
7: $j \leftarrow i$
8: **returnează** A

Operaţiile realizate de algoritm pot fi vizualizate în figura 2.1. în mulţimea A se introduc activităţile selectate. Variabila j identifică ultima activitate introdusă în A. Deoarece activităţile sunt considerate în ordinea nedescrescătoare a timpilor lor de terminare, f_j, va reprezenta întotdeauna timpul maxim de terminare a oricărei activităţi din A. Aceasta înseamnă că

$$f_j = \max\{f_k : \in A\} \qquad (2.2)$$

În liniile 2-3 din algoritm se selectează activitatea 1, se iniţializează A astfel încât să nu conţină decât această activitate, iar variabila j ia ca valoare această activitate. În continuare liniile 4-7 consideră pe rând fiecare activitate i şi o adaugă mulţimii A dacă este compatibilă cu toate celelalte activităţi deja selectate. Pentru a vedea dacă activitatea i este compatibilă cu toate celelalte activităţi existente la momentul curent în A, este suficient, conform formulei (2.2), să fie îndeplinită condiţia din linia 5 adică momentul de pornire s_i, să nu fie mai devreme decât momentul de oprire f_j, al activităţii cel mai recent adăugate mulţimii A. Dacă activitatea i este compatibilă, atunci în liniile 6-7 ea este adăugată mulţimii A, iar variabila j este actualizată. Procedura **SELECTOR-ACTIVITĂŢI-GREEDY** este foarte eficientă. Ea poate planifica o mulţime S de n activităţi în $\Theta(n)$, presupunând că activităţile au fost deja ordonate după timpul lor de terminare. Activitatea aleasă de procedura **SELECTOR-ACTIVITĂŢI-**

GREEDY este întotdeauna cea cu primul timp de terminare care poate fi planificată legal. Activitatea astfel selectată este o alegere "greedy" (lacomă) în sensul că, intuitiv, ea lasă posibilitatea celorlalte activităţi rămase pentru a fi planificate. Cu alte cuvinte, alegerea greedy maximizează cantitatea de timp neplanificată rămasă.

i	s_i	f_i
1	1	4
2	3	5
3	0	6
4	5	7
5	3	8
6	5	9
7	6	10
8	8	11
9	8	12
10	2	13
11	12	14

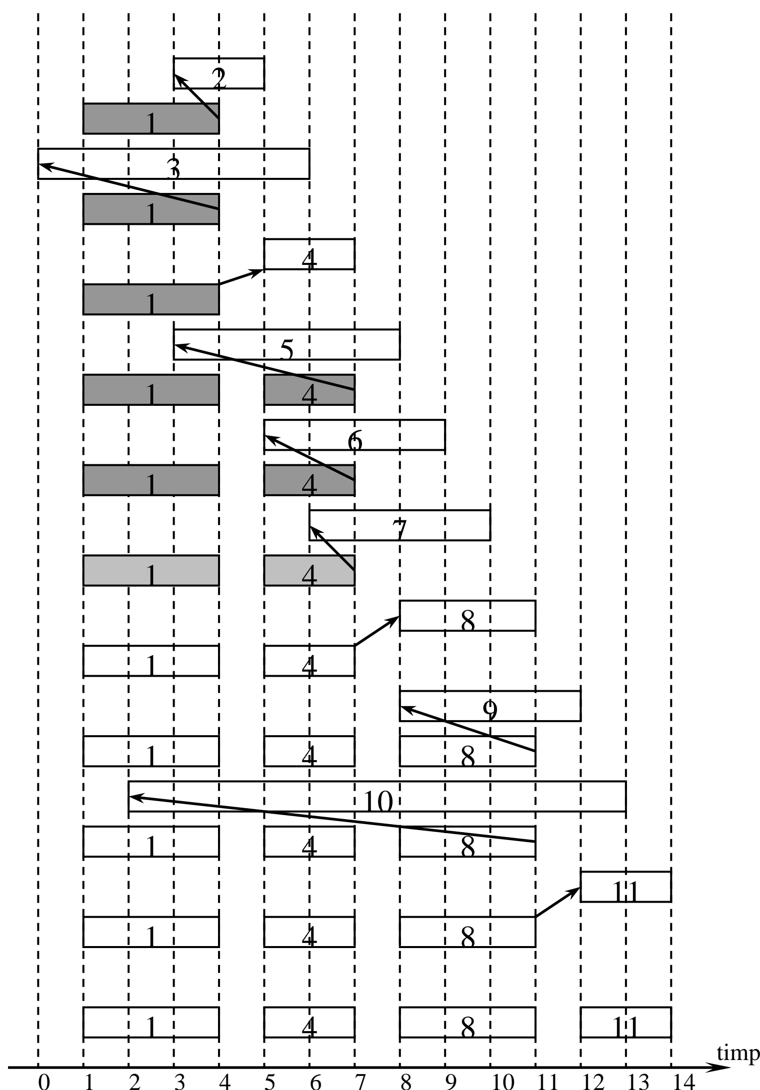


Figura 2.1. Operaţiile algoritmului **SELECTOR-ACTIVITĂŢI-GREEDY** asupra celor 11 activităţi date în stânga. Fiecare linie a figurii corespunde unei iteraţii din ciclul pentru din liniile 4-7. Activităţile care au fost selectate pentru a fi incluse în mulţimea *A* sunt haşurate, iar activitatea curentă *i* este nehaşurată. Dacă timpul de pornire s_i, al activităţii i este mai mic decât timpul de terminare al activităţii *j* (săgeata dintre ele este spre stânga), activitatea este ignorată, în caz contrar (săgeata este îndreptată spre dreapta), activitatea este acceptată şi este adăugată mulţimii *A*.

2.1.1. Demonstrarea corectitudinii algoritmului greedy

Algoritmii de tip greedy nu furnizează întotdeauna soluţiile optime. Cu toate acestea, algoritmul **SELECTOR-ACTIVITĂŢI-GREEDY** determină întotdeauna o soluţie optimă pentru o instanţă a problemei selectării activităţilor.

Teorema 2.1. Algoritmul **SELECTOR-ACTIVITĂŢI-GREEDY** furnizează soluţii de dimensiune maximă pentru problema selectării activităţilor.

Demonstraţie. Fie $S = \{1,2,..., n\}$ mulţimea activităţilor care trebuie planificate. Deoarece presupunem că activităţile sunt ordonate după timpul de terminare, activitatea 1 se va termina cel mai devreme. Vrem să arătăm că există o soluţie optimă care începe cu activitatea 1, conform unei alegeri greedy.

Să presupunem că $A \subseteq S$ este o soluţie optimă pentru o instanţă dată a problemei selectării activităţilor. Vom ordona activităţile din *A* după timpul de terminare. Mai presupunem că prima activitate din *A* este activitatea *k*. Dacă $k = 1$, planificarea mulţimii *A* începe cu o alegere greedy. Dacă $k \neq 1$ vrem să arătăm că există o altă soluţie optimă *B* a lui *S* care începe conform alegerii greedy, cu activitatea 1. Fie $B = A - \{k\} \cup U \{1\}$. Deoarece $f_1 \leq f_k$,

activităţile din B sunt distincte, şi cum B are acelaşi număr de activităţi ca şi A, B este de asemenea optimă. Deci B este o soluţie optimă pentru S, care conţine alegerea greedy a activităţii 1. Am arătat astfel că există întotdeauna o planificare optimă care începe cu o alegere greedy.

Mai mult, o dată ce este făcută alegerea greedy a activităţii 1, problema se reduce la determinarea soluţiei optime pentru problema selectării acelor activităţi din S care sunt compatibile cu activitatea 1. Aceasta înseamnă că dacă A este o soluţie optimă pentru problema iniţială, atunci $A' = A - \{1\}$ este o soluţie optimă pentru problema selectării activităţilor

$S' = \{i \in S, s_i \geq f_1\}$. De ce? Dacă am putea găsi o soluţie B' pentru S' cu mai multe activităţi decât A', adăugarea activităţii 1 la B' va conduce la o soluţie B a lui S cu mai multe activităţi decât A, contrazicându-se astfel optimalitatea mulţimii A. în acest mod, după ce este făcută fiecare alegere greedy, ceea ce rămâne este o problemă de optimizare de aceeaşi formă ca problema iniţială. Prin inducţie după numărul de alegeri făcute, se arată că realizând o alegere greedy la fiecare pas, se obţine o soluţie optimă.

2.2. Coduri Huffman

Codurile Huffman reprezintă o tehnică foarte utilizată şi eficientă pentru compactarea datelor; în funcţie de caracteristicile fişierului care trebuie comprimat, spaţiul economisit este între 20% şi 90%. Algoritmul greedy pentru realizarea acestei codificări utilizează un tabel cu frecvenţele de apariţie ale fiecărui caracter. Ideea este de a utiliza o modalitate optimă pentru reprezentarea fiecărui caracter sub forma unui şir binar.

Sa presupunem că avem un fişier ce conţine 100.000 de caractere, pe care dorim să îl memorăm într-o formă compactată. Frecvenţele de apariţie ale caracterelor în text sunt date de figura 2.2: există doar şase caractere diferite şi fiecare dintre ele apare de 45.000 de ori.

	a	b	c	d	e	f
Frecvenţa(în mii)	45	13	12	16	9	5
Cuvânt codificat; lungime fixă	000	001	010	011	100	101
Cuvânt codificat; lungime variabilă	0	101	100	111	1101	1100

Figura 2.2. O problemă de codificare a caracterelor. Un fişier de 100.000 de caractere conţine doar caracterele `a-f`, cu frecvenţele indicate. Dacă fiecărui caracter îi este asociat un cuvânt de cod pe 3 biţi, fişierul codificat va avea 300.000 de biţi. Folosind codificarea cu lungime variabilă, fişierul codificat va ocupa doar 224.000 de biţi.

Exista mai multe modalităţi de reprezentare a unui astfel de fişier. Vom considera problema proiectării unui cod *binar al caracterelor* (pe scurt cod) astfel încât fiecare caracter este reprezentat printr-un şi binar unic. Dacă utilizăm un *cod de lungime fixă,* avem nevoie de 3 biţi pentru a reprezenta şase caractere: `a=000,b=001, ...,f=101`. Această metodă necesită 300.000 de biţi pentru a codifica tot fişierul. Se pune problema dacă se poate face o compactare şi mai bună.

O *codificare cu lungime variabilă* poate îmbunătăţi semnificativ performanţele, atribuind caracterelor cu frecvenţe mai mari cuvinte de cod mai scurte iar celor cu frecvenţe mai reduse cuvinte de cod mai lungi. Figura 2.3(b) prezintă o astfel de codificare; şirul 0 având lungimea de 1 bit reprezintă caracterul `a` in timp ce şirul 1100 de lungime 4 reprezintă caracterul `f`. Această codificare necesită (45 • 1 + 13 • 3 4-12 • 3 + 16 • 3 + 9 • 4 + 5 • 4) • 1.000 = 224.000 biţi pentru a reprezenta un fişier şi economiseşte aproximativ 25% din spaţiu. Vom vedea că aceasta este de fapt o codificare-caracter optimă pentru acest fişier.

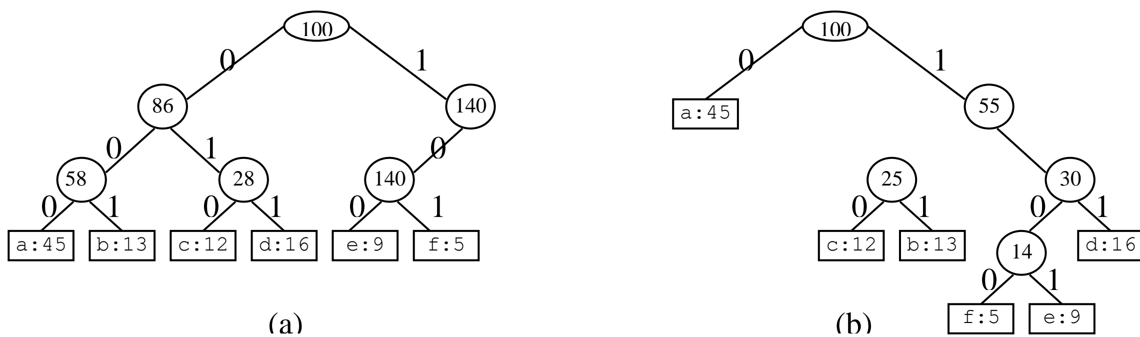


Figura 2.3. Arborii corespunzători schemelor de codificare din figura 2.2. Fiecare frunză este etichetată cu un caracter şi cu frecvenţa de apariţie a acestuia. Fiecare nod intern este etichetat cu suma ponderilor frunzelor din

subarborele aferent. (a) Arborele corespunzător codificării de lungime fixă `a=000,...,f=101`. (b) Arborele asociat codificării prefix optime `a=0,b=101,...,f=1100`.

2.2.1. Coduri prefix

Vom considera în continuare doar codificările în care nici un cuvânt de cod nu este prefixul altui cuvânt. Astfel de codificări se numesc *codificări prefix.* Se poate arăta că o compresie optimă a datelor, realizată prin codificarea caracterelor, poate fi realizată şi prin codificarea prefix, deci considerarea codificării prefix nu scade din generalitate.

Codificarea prefix este utilă deoarece simplifică atât codificarea (deci compactarea) cât şi decodificarea. Codificarea este întotdeauna simplă pentru orice codificare binară a caracterelor; se concatenează cuvintele de cod reprezentând fiecare caracter al fişierului. De exemplu, prin codificarea prefix având lungime variabila din figura 17.3, putem codifica un fişier de 3 caractere `abc` astfel: 0•101•100=0101100, unde semnul "•" reprezintă operaţia de concatenare.

Decodificarea este de asemenea relativ simplă pentru codurile prefix. Cum nici un cuvânt de cod nu este prefixul altuia. Începutul oricărui fişier codificat, nu este ambiguu. Putem deci sa identificăm cuvântul iniţial de cod, să îl "traducem" in caracterul original, să-l îndepărtăm din fişierul codificat şi să repetăm procesul pentru fişierul codificat rămas. În exemplul nostru, şirul 001011101 se translatează automat în 0•0•101•1101, secvenţă care se decodifică în `aabe`.

Procesul de decodificare necesită o reprezentare convenabilă a codificării prefix astfel încât cuvântul iniţial de cod sa poată fi uşor identificat. O astfel de reprezentare poate fi dată de un arbore binar ale cărui frunze sunt caracterele date. Interpretăm un cuvânt de cod binar pentru un caracter ca fiind drumul de la rădăcina până la caracterul respectiv, unde 0 reprezintă "mergi la fiul stâng" iar 1 "mergi la fiul drept". În figura 3 sunt prezentaţi arborii pentru cele două codificări ale exemplului nostru. Observaţi că aceştia nu sunt arbori binari de căutare. deoarece frunzele nu trebuie să fie ordonate, iar nodurile interne nu conţin chei pentru caractere.

O codificare optimă pentru un fişier este întotdeauna reprezentată printr-un arbore binar complet, în care fiecare vârf care nu este frunză are doi fii.

Codificarea cu lungime fixă din exemplul de mai sus nu este optimă deoarece arborele asociat, prezentat în figura 3(a), nu este un arbore binar complet: există două cuvinte de cod care încep cu 10..., dar nici unul care să înceapă cu 11.... Conform celor de mai sus ne putem restrânge atenţia numai asupra arborilor binari compleţi, deci dacă C este alfabetul din care fac parte caracterele, atunci arborele pentru o codificare prefix optimă are exact $|C|$ frunze, una pentru fiecare literă din alfabet, şi exact $|C| - 1$ noduri interne.

Dându-se un arbore T, corespunzător unei codificări prefix, este foarte simplu să calculăm numărul de biţi necesari pentru a codifica un fişier. Pentru fiecare caracter c din alfabet, fie $f(c)$ frecvenţa lui c în fişier şi să notăm cu $d_T(c)$ adâncimea frunzei în arbore (adică nivelul pe care se află). Să observăm că $d_T(c)$ reprezintă de asemenea cuvântul de cod pentru caracterul c. Numărul de biţi necesari pentru a codifica un fişier este

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

Vom numi acest număr *costul* arborelui r.

2.2.2. Construcţia unui cod Huffman

Huffman a inventat un algoritm greedy care construieşte o codificare prefix optimă numită *codul Huffman.* Algoritmul construieşte arborele corespunzător codificării optime, într-o manieră "bottom-up". Se începe cu o mulţime de $|C|$ frunze şi se realizează o secvenţă de $|C|-1$ operaţii de interclasare pentru a crea arborele final.

În algoritmul care urmează, vom presupune că C este o mulţime de n caractere şi fiecare caracter $c \in C$ este un obiect având o frecvenţă dată $f[c]$. Va fi folosită o coadă de priorităţi pentru a identifica cele două obiecte cu frecvenţa cea mai redusă care vor fuziona. Rezultatul interclasării celor două obiecte este un nou obiect a cărui frecvenţă este suma frecvenţelor celor două obiecte care au fost interclasate.

HUFFMAN (C)
1: $n \leftarrow |C|$
2: $Q \leftarrow C$
3: **pentru** $i \leftarrow 1, n-1$ **execută**
4: $z \leftarrow$ ALOCĂ-NOD()
5: $x \leftarrow$ *stânga*$[z] \leftarrow$ EXTRAGE+MIN(Q)
6: $y \leftarrow$ dreapta$[z] \leftarrow$ EXTRAGE-MIN(Q)
7: $f[z] \leftarrow f[x]+f[y]$
8: INSEREAZĂ (Q,3)
9: **returnează** EXTRAGE-MIN(Q)

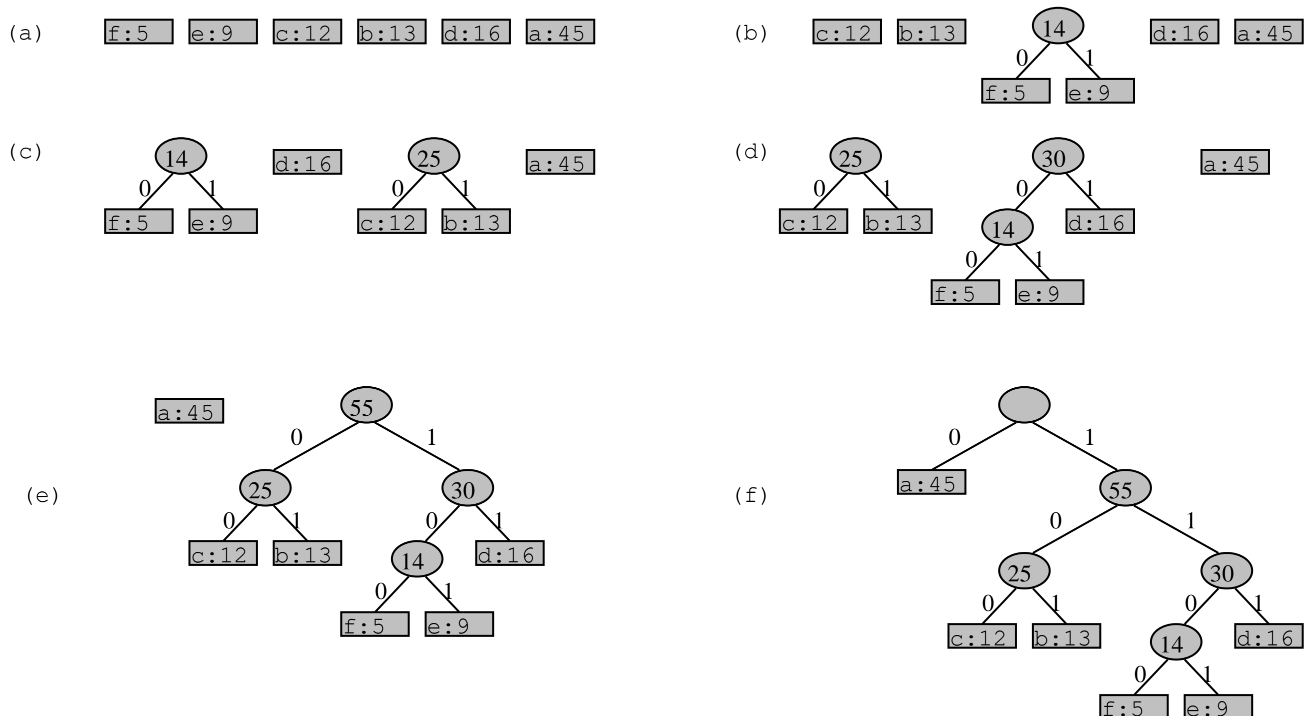


Figura 2.4 Paşii algoritmului Huffman pentru frecvenţele date în figura 2.2. Fiecare parte ilustrează conţinutul cozii ordonate crescător după frecvenţă. La fiecare pas, cei doi arbori cu cele mai scăzute frecvenţe fuzionează. Frunzele figurează ca dreptunghiuri ce conţin un caracter şi frecvenţa sa. Nodurile interne figurează ca cercuri ce conţin suma frecvenţelor fiilor lor. O muchie care leagă un nod intern cu fiii ei este etichetată cu 0 dacă este o muchie către un fiu stâng, respectiv cu 1 dacă este câtre fiul drept. Cuvântul de cod pentru o literă este secvenţa etichetelor de pe muchiile care leagă rădăcina de frunza asociată caracterului respectiv. (a) Mulţimea iniţială de n=6 noduri, unul pentru fiecare literă. (b)-(e) Etape intermediare. (f) Arborele final.

Pentru exemplul nostru, algoritmul lui Huffman lucrează ca în figura 2.3. Deoarece există 6 litere în alfabet, dimensiunea iniţială a cozii este n = 6, iar pentru a construi arborele sunt necesari 5 paşi de interclasare. Arborele final reprezintă codificarea prefix optimă. Codificarea unei litere este secvenţa etichetelor nodurilor ce formează drumul de la rădăcină la literă.

În linia 2 coada de priorităţi Q este iniţializată cu caracterele din C. Ciclul ***for*** din liniile 3-8 extrage în mod repetat două noduri x şi y cu cea mai mică frecvenţă din coadă, şi le înlocuieşte în coadă cu un nou nod z, reprezentând fuziunea lor. Frecvenţa lui z este calculată în linia 7 ca fiind suma frecvenţelor lui x şi y. Nodul z are pe x ca fiu stâng şi pe y ca fiu drept. (Această ordine este arbitrară; interschimbarea fiilor stâng şi drept ai oricărui nod conduce la o codificare diferită, dar de acelaşi cost.) După n - 1 fuzionări, unicul nod rămas în coadă, rădăcina arborelui de codificare, este returnat în linia 9.

2.2.3. Corectitudinea algoritmului lui Huffman

Pentru a demonstra că algoritmul de tip greedy al lui Huffman este corect, vom arăta că problema determinării unei codificări prefix optime implică alegeri greedy şi are o substructură optimă. Următoarea lemă se referă la proprietatea alegerii greedy.

Lema 2.1. Fie C un alfabet în care fiecare caracter $c \in C$ are frecvenţa $f[c]$. Fie x şi y două caractere din C având cele mai mici frecvenţe. Atunci există o codificare prefix optimă pentru C în care cuvintele de cod pentru x şi y au aceeaşi lungime şi diferă doar pe ultimul bit.

Demonstraţie. Ideea demonstraţiei este de a lua arborele T reprezentând o codificare prefix optimă şi a-l modifica pentru a realiza un arbore reprezentând o altă codificare prefix optimă. În noul arbore, caracterele x şi y vor apare ca frunze cu acelaşi tată şi se vor afla pe nivelul maxim în arbore. Dacă putem realiza acest lucru, atunci cuvintele lor de cod vor avea aceeaşi lungime şi vor diferi doar pe ultimul bit.

Fie b şi c două caractere reprezentând noduri terminale (frunze) situate pe nivelul maxim al arborelui T. Fără a restrânge generalitatea, vom presupune că $f[b] \le f[c]$ şi $f[x] \le f[y]$. Cum $f[x]$ şi $f[y]$ sunt frunzele cu frecvenţele cele mai scăzute, in această ordine, iar f[b] şi $f[c]$ sunt deci frecvenţe arbitrare, în ordine, avem $f[x] \le f[b]$ şi $f[y] \le f[c]$. După cum se vede in figura 2.5, vom schimba în T poziţiile lui b şi x pentru a produce arborele T', iar apoi vom schimba în T' poziţiile lui c şi y pentru a obţine arborele T''. Conform ecuaţiei (17.3), diferenţa costurilor lui T şi T' este

$$\begin{aligned}B(T) - B(T') &= \sum_{c \in C} f(c) d_T(c) - \sum_{c \in C} f(c) d_{T'}(c) \\ &= f[x]d_T(x) + f[b]d_T(b) - f[x]d_{T'}(x) - f[b]d_{T'}(b) \\ &= f[x]d_T(x) + f[b]d_T(b) - f[x]d_{T'}(b) - f[b]d_{T'}(x) \\ &= (f[b] - f[x])(d_T(b) - d_T(x)) \ge 0,\end{aligned}$$

deoarece atât $f[b]$ - $f[x]$ şi $d_T[b]$ -$d_T[x]$ sunt nenegative. Mai precis, $f[b]$ - $f[x]$ este nenegativ deoarece x este frunză având frecvenţa minimă iar $d_T[b]$ -$d_T[x]$ este nenegativ pentru că b este o frunză aflată pe nivelul maxim în T. În mod analog, deoarece interschimbarea lui y cu c nu măreşte costul, diferenţa $B(T) \le B(T'')$ este nenegativă. Astfel $B(T) \le B(T'')$, ceea ce implică $B(T'') = B(T)$. Aşadar T'' este un arbore optim în care x şi y apar ca noduri terminale fraţi, şi se află pe nivelul maxim, ceea ce trebuia demonstrat.

Din lema 2.1 se deduce faptul ca procesul de construcţie a unui arbore optim prin fuzionări poate, fără a restrânge generalitatea. să înceapă cu o alegere greedy a interclasării acelor două caractere cu cea mai redusă frecvenţă. De ce este aceasta o alegere greedy? Putem interpreta costul unei singure interclasări ca fiind suma frecvenţelor celor două obiecte care fuzionează. La fiecare pas, dintre toate interclasările posibile, algoritmul HUFFMAN o alege pe aceea care determină costul minim.

Lema 2.2. Fie T un arbore binar complet reprezentând o codificare prefix optimă peste un alfabet C, unde frecvenţa $f[c]$ este definită pentru fiecare caracter $c \in C$. Considerăm două caractere x şi y oarecare care apar ca noduri terminale fraţi în T, şi fie z tatăl lor. Atunci, considerând z ca un caracter având frecvenţa $f[z] = f[x] + f[y]$, arborele $T' = T$ - $\{x,y\}$ reprezintă o codificare prefix optimă pentru alfabetul $C' = C - \{x,y\} \cup \{z\}$.

Demonstraţie. Vom arăta mai întâi că $B(T)$, costul arborelui T, poate fi exprimat în funcţie de costul $B(T')$ al arborelui T' considerând costurile componente din ecuaţia (17.3). Pentru fiecare $c \in C$ - $\{x,y\}$, avem $d_T[c] = d_{T'}[c]$, şi $f[c]\, d_t[c] = f[c]\, d_{t'}[c]$. Cum $d_T[x] = d_T[y] = d_{T'}[z] + 1$, avem

$$f[x]d_T(x) + f[y]d_T(y) = f[x] + f[y])d_{T'}(z) + 1) = f[z]d_{T'}(z) + (f[x] + f[y]),$$

de unde deducem că

$$B(T) = B(T') + f[x] + f[y].$$

Dacă T' reprezintă o codificare prefix care nu este optimă pentru alfabetul C', atunci există un arbore T'' ale cărui frunze sunt caractere în C′ astfel încât $B(T'') < B(T')$. Cum z este tratat ca un caracter în C', el va apare ca frunză în T''. Dacă adăugăm x şi y ca fiind fiii lui z in T'', atunci vom obţine o codificare prefix pentru C având costul $B(T'') + f[x] + f[y] < B(T)$, ceea ce intră în contradicţie cu optimalitatea lui T. Deci T' trebuie să fie optim pentru alfabetul C'.

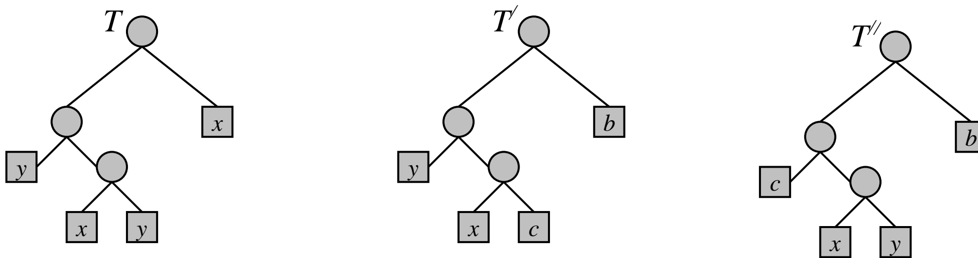


Figura 2.5 O ilustrare a pasului cheie din demonstraţia lemei 2.1. În arborele optim T, b şi c sunt două dintre frunzele aflate pe cel mai de jos nivel în arbore; aceste noduri se consideră fraţi. x şi y sunt două frunze pe care algoritmul Huffman le interclasează primele; ele apar în poziţii arbitrare în T. Frunzele b şi x sunt interschimbate pentru a obţine arborele T'. Apoi, frunzele c şi y sunt interschimbate pentru a obţine arborele T''. Cum cele două interschimbări nu măresc costul, arborele rezultat T'' este de asemenea un arbore optim.

2.3. Arbori parţiali de cost minim

Fie $G = <V, M>$ un graf neorientat conex, unde V este mulţimea vârfurilor si M este mulţimea muchiilor. Fiecare muchie are un *cost* nenegativ (sau o *lungime* nenegativă). Problema este sa găsim o submulţime $A \subseteq M$, astfel încât toate vârfurile din V sa rămână conectate atunci când sunt folosite doar muchii din A, iar suma

lungimilor muchiilor din A sa fie cat mai mica. Căutam deci o submulţime A de cost total minim. Aceasta problema se mai numeşte si *problema conectării oraşelor cu cost minim*, având numeroase aplicaţii.

Graful parţial $<V, A>$ este un arbore si este *numit arborele parţial de cost minim* al grafului G (*minimal spanning tree*). Un graf poate avea mai mulţi arbori parţiali de cost minim si acest lucru se poate verifica pe un exemplu.

Vom prezenta doi algoritmi greedy care determina arborele parţial de cost minim al unui graf. In terminologia metodei greedy, vom spune ca o mulţime de muchii este o *soluţie*, daca constituie un arbore parţial al grafului G, si este *fezabila*, daca nu conţine cicluri. O mulţime fezabilă de muchii este *promiţătoare*, daca poate fi completata pentru a forma soluţia optima. O muchie *atinge* o mulţime data de vârfuri, daca exact un capăt al muchiei este in mulţime. Următoarea proprietate va fi folosita pentru a demonstra corectitudinea celor doi algoritmi.

Proprietatea 2.1 Fie $G = <V, M>$ un graf neorientat conex in care fiecare muchie are un cost nenegativ. Fie $W \subset V$ o submulţime stricta a vârfurilor lui G si fie $A \subseteq M$ o mulţime promiţătoare de muchii, astfel încât nici o muchie din A nu atinge W. Fie m muchia de cost minim care atinge W. Atunci, $A \cup \{m\}$ este promiţătoare.

Demonstraţie: Fie B un arbore parţial de cost minim al lui G, astfel încât $A \subseteq B$ (adică, muchiile din A sunt conţinute in arborele B). Un astfel de B trebuie sa existe, deoarece A este promiţătoare. Daca $m \in B$, nu mai rămâne nimic de demonstrat. Presupunem ca $m \notin B$. Adăugându-l pe m la B, obţinem exact un ciclu (Exercitiul3.2). In acest ciclu, deoarece m atinge W, trebuie sa mai existe cel puţin o muchie m' care atinge si ea pe W (altfel, ciclul nu se închide). Eliminându-l pe m', ciclul dispare si obţinem un nou arbore parţial B' al lui G. Costul lui m este mai mic sau egal cu costul lui m', deci costul total al lui B' este mai mic sau egal cu costul total al lui B. De aceea, B' este si el un arbore parţial de cost minim al lui G, care include pe m. Observam ca $A \subseteq B'$ deoarece muchia m', care atinge W, nu poate fi in A. Deci, $A \cup \{m\}$ este promiţătoare.

Mulţimea iniţiala a candidaţilor este M. Cei doi algoritmi greedy aleg muchiile una cate una intr-o anumita ordine, aceasta ordine fiind specifica fiecărui algoritm.

2.3.1. Algoritmul lui Kruskal

Arborele parţial de cost minim poate fi construit muchie, cu muchie, după următoarea metoda a lui Kruskal (1956): se alege întâi muchia de cost minim, iar apoi se adaugă repetat muchia de cost minim nealeasă anterior si care nu formează cu precedentele un ciclu. Alegem astfel $\#V-1$ muchii. Este uşor de dedus ca obţinem in final un arbore. Este însă acesta chiar arborele parţial de cost minim căutat?

Înainte de a răspunde la întrebare, sa consideram, de exemplu, graful din Figura 2.6(a). Ordonam crescător (in funcţie de cost) muchiile grafului: {1, 2}, {2, 3}, {4, 5}, {6, 7}, {1, 4}, {2, 5}, {4, 7}, {3, 5}, {2, 4}, {3, 6}, {5, 7}, {5, 6} si apoi aplicam algoritmul. Structura componentelor conexe este ilustrata, pentru fiecare pas, in Figura 2.7.

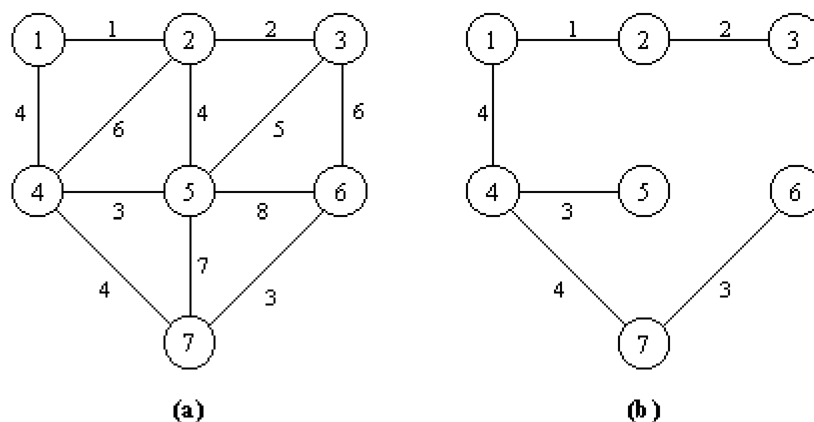


Figura 2.6 Un graf si arborele său parţial de cost minim.

Pasul	Muchia considerata	Componentele conexe ale subgrafului <*V*, *A*>
iniţializare	—	{1}, {2}, {3}, {4}, {5}, {6}, {7}
1	{1, 2}	{1, 2}, {3}, {4}, {5}, {6}, {7}
2	{2, 3}	{1, 2, 3}, {4}, {5}, {6}, {7}
3	{4, 5}	{1, 2, 3}, {4, 5}, {6}, {7}
4	{6, 7}	{1, 2, 3}, {4, 5}, {6, 7}
5	{1, 4}	{1, 2, 3, 4, 5}, {6, 7}
6	{2, 5}	respinsa (formează ciclu)
7	{4, 7}	{1, 2, 3, 4, 5, 6, 7}

Figura 2.7 Algoritmul lui Kruskal aplicat grafului din Figura 2.6(a)

Mulţimea *A* este iniţial vida şi se completează pe parcurs cu muchii acceptate (care nu formează un ciclu cu muchiile deja existente in *A*). In final, mulţimea *A* va conţine muchiile {1, 2}, {2, 3}, {4, 5}, {6, 7}, {1, 4}, {4, 7}. La fiecare pas, graful parţial <*V*, *A*> formează o pădure de componente conexe, obţinută din pădurea precedenta unind două componente. Fiecare componentă conexă este la rândul ei un arbore parţial de cost minim pentru vârfurile pe care le conectează. Iniţial, fiecare vârf formează o componenta conexa. La sfârşit, vom avea o singură componentă conexă, care este arborele parţial de cost minim căutat (Figura 2.6(b)).

Ceea ce am observat în acest caz particular este valabil şi pentru cazul general, din Proprietatea 2.1 rezultând:

Proprietatea 2.2 În algoritmul lui Kruskal, la fiecare pas, graful parţial <*V*, *A*> formează o pădure de componente conexe, în care fiecare componentă conexă este la rândul ei un arbore parţial de cost minim pentru vârfurile pe care le conectează. In final, se obţine arborele parţial de cost minim al grafului *G*.

Pentru a implementa algoritmul, trebuie să putem manipula submulţimile formate din vârfurile componentelor conexe. Folosim pentru aceasta o structură de mulţimi disjuncte si procedurile de tip *find* si *merge* (Vezi disciplina „Structuri de date şi algoritmi”). In acest caz, este preferabil să reprezentam graful ca o listă de muchii cu costul asociat lor, astfel încât să putem ordona această listă în funcţie de cost. Iată algoritmul:

```
KRUSCAL(G = <V, M>)
1. {iniţializare}
2. sortează M crescător in funcţie de cost
3. n ← #V
4. A ← ∅ {va conţine muchiile arborelui parţial de cost minim}
5. {iniţializează n mulţimi disjuncte conţinând fiecare cate un element din V}
   {bucla greedy}
repeat
   {u, v} ← muchia de cost minim care
            încă nu a fost considerata
   ucomp ← find(u)
   vcomp ← find(v)
   if ucomp ≠ vcomp    then   merge(ucomp, vcomp)
                              A ← A ∪ {{u, v}}
until #A = n-1
return A
```

Pentru un graf cu *n* vârfuri si *m* muchii, presupunând ca se folosesc procedurile *find*3 si *merge*3, numărul de operaţii pentru cazul cel mai nefavorabil este in:

- $O(m \log m)$ pentru a sorta muchiile. Deoarece $m \leq n(n-1)/2$, rezulta $O(m \log m) \subseteq O(m \log n)$. Mai mult, graful fiind conex, din $n-1 \leq m$ rezulta si $O(m \log n) \subseteq O(m \log m)$, deci $O(m \log m) = O(m \log n)$.
- $O(n)$ pentru a iniţializa cele *n* mulţimi disjuncte.
- Cele cel mult 2*m* operaţii *find*3 si $n-1$ operaţii *merge*3 necesita un timp in $O((2m+n-1) \lg^* n)$, după cum am specificat in Capitolul 3. Deoarece $O(\lg^* n) \subseteq O(\log n)$ si $n-1 \leq m$, acest timp este si in $O(m \log n)$.
- $O(m)$ pentru restul operaţiilor.

Deci, pentru cazul cel mai nefavorabil, algoritmul lui Kruskal necesita un timp in $O(m \log n)$.

O alta varianta este sa păstram muchiile intr-un min-heap. Obţinem astfel un nou algoritm, in care iniţializarea se face intr-un timp in $O(m)$, iar fiecare din cele $n-1$ extrageri ale unei muchii minime se face intr-un timp in $O(\log m) = O(\log n)$. Pentru cazul cel mai nefavorabil, ordinul timpului rămâne acelaşi cu cel al vechiului algoritm.

Avantajul folosirii min-heap-ului apare atunci când arborele parţial de cost minim este găsit destul de repede si un număr considerabil de muchii rămân netestate. In astfel de situaţii, algoritmul vechi pierde timp, sortând in mod inutil si aceste muchii.

5.2.2 Algoritmul lui Prim

Cel de-al doilea algoritm greedy pentru determinarea arborelui parţial de cost minim al unui graf se datorează lui Prim (1957). In acest algoritm, la fiecare pas, mulţimea A de muchii alese împreuna cu mulţimea U a vârfurilor pe care le conectează formează un arbore parţial de cost minim pentru subgraful $<U, A>$ al lui G. Iniţial, mulţimea U a vârfurilor acestui arbore conţine un singur vârf oarecare din V, care va fi rădăcina, iar mulţimea A a muchiilor este vida. La fiecare pas, se alege o muchie de cost minim, care se adaugă la arborele precedent, dând naştere unui nou arbore parţial de cost minim (deci, exact una dintre extremitatile acestei muchii este un vârf in arborele precedent). Arborele parţial de cost minim creste "natural", cu cate o ramura, până când va atinge toate vârfurile din V, adică până când $U = V$. Funcţionarea algoritmului, pentru exemplul din Figura 6(a), este ilustrata in Tabelul 6.2. La sfârşit, A va conţine aceleaşi muchii ca si in cazul algoritmului lui Kruskal. Faptul ca algoritmul funcţionează întotdeauna corect este exprimat de următoarea proprietate, pe care o puteţi demonstra folosind Proprietatea 6.2.

Pasul	**Muchia considerata**	***U***
iniţializare	—	{1}
1	{2, 1}	{1, 2}
2	{3, 2}	{1, 2, 3}
3	{4, 1}	{1, 2, 3, 4}
4	{5, 4}	{1, 2, 3, 4, 5}
5	{7, 4}	{1, 2, 3, 4, 5, 6}
6	{6, 7}	{1, 2, 3, 4, 5, 6, 7}

Tabelul 5.2 Algoritmul lui Prim aplicat grafului din Figura 6.4a.

Proprietatea 5.4 In algoritmul lui Prim, la fiecare pas, $<U, A>$ formează un arbore parţial de cost minim pentru subgraful $<U, A>$ al lui G. In final, se obţine arborele parţial de cost minim al grafului G.

Descrierea formala a algoritmului este data in continuare.

```
function Prim-formal(G = <V, M>)
{iniţializare}
A ← ∅   {va conţine muchiile arborelui parţial de cost minim}
U ← {un vârf oarecare din V}
{bucla greedy}
while U ≠ V do
    găseşte {u, v} de cost minim astfel ca u ∈ V \ U si v ∈ U
    A ← A ∪ {{u, v}}
    U ← U ∪ {u}
return A
```

Pentru a obţine o implementare simpla, presupunem ca: vârfurile din V sunt numerotate de la 1 la n, $V = \{1, 2, ..., n\}$; matricea simetrica C da costul fiecărei muchii, cu $C[i, j] = +\infty$, daca muchia $\{i, j\}$ nu exista. Folosim doua tablouri paralele. Pentru fiecare $i \in V \setminus U$, *vecin*[*i*] conţine vârful din U, care este conectat de i printr-o muchie de cost minim; *mincost*[*i*] da acest cost. Pentru $i \in U$, punem *mincost*[*i*] $= -1$. Mulţimea U, in mod arbitrar iniţializată cu {1}, nu este reprezentata explicit. Elementele *vecin*[1] si *mincost*[1] nu se folosesc.

```
function Prim(C[1 .. n, 1 .. n])
{iniţializare; numai vârful 1 este in U}
A ← ∅
for i ← 2 to n do  vecin[i]   ← 1
                   mincost[i] ← C[i, 1]
{bucla greedy}
repeat n–1 times
    min ← +∞
    for j ← 2 to n do
        if 0 < mincost[ j] < min  then  min ← mincost[ j]
```

```
                                   k ← j
   A ← A ∪ {{k, vecin[k]}}
   mincost[k] ← –1  {adaugă vârful k la U}
   for j ← 2 to n do
      if C[k, j] < mincost[ j] then    mincost[ j] ← C[k, j]
                                       vecin[ j] ← k
return A
```

Bucla principala se executa de n–1 ori si, la fiecare iteraţie, buclele **for** din interior necesita un timp in $O(n)$. Algoritmul *Prim* necesita, deci, un timp in $O(n^2)$. Am văzut ca timpul pentru algoritmul lui Kruskal este in $O(m \log n)$, unde $m = \#M$. Pentru un graf *dens* (adică, cu foarte multe muchii), se deduce ca m se apropie de $n(n–1)/2$. In acest caz, algoritmul *Kruskal* necesita un timp in $O(n^2 \log n)$ si algoritmul *Prim* este probabil mai bun. Pentru un graf *rar* (adică, cu un număr foarte mic de muchii), m se apropie de n si algoritmul *Kruskal* necesita un timp in $O(n \log n)$, fiind probabil mai eficient decât algoritmul *Prim*.

6. PROGRAMAREA DINAMICĂ

Programarea dinamică, asemenea metodelor *divide şi stăpâneşte,* rezolvă problemele combinând soluţiile unor subprobleme. (în acest context, termenul de "programare" se referă la o metodă tabelară şi nu la scrierea codului pentru un calculator.) După cum am arătat mai sus, algoritmii *divide şi stăpâneşte* partiţionează problema în subprobleme independente, rezolvă recursiv subproblemele şi apoi combina soluţiile lor pentru a rezolva problema iniţială. Spre deosebire de această abordare, programarea dinamică este aplicabilă atunci când subproblemele nu sunt independente, adică subproblemele au în comun sub-subprobleme. În acest context, un algoritm de tipul *divide şi stăpâneşte* ar presupune mai multe calcule decât ar fi necesar dacă s-ar rezolva în mod repetat sub-subproblemele comune. Un algoritm bazat pe programare dinamică rezolvă fiecare sub-subproblemă o singură dată şi memorează soluţia acesteia într-un tablou, prin aceasta evitând recalcularea soluţiei ori de câte ori respectiva sub-subproblemă apare din nou.

În general, metoda programării dinamice se aplică ***problemelor de optimizare***. Asemenea probleme pot avea mai multe soluţii posibile. Fiecare soluţie are o valoare iar ceea ce se doreşte este determinarea soluţiei a cărei valoare este optimă (minimă sau maximă). O asemenea soluţie se numeşte o *soluţie* optimă a problemei prin contrast cu *soluţia* optimă, deoarece pot fi mai multe soluţii care realizează valoarea optimă.

Dezvoltarea unui algoritm bazat pe programarea dinamică poate fi împărţită într-o secvenţă de patru paşi.

1. Caracterizarea structurii unei soluţii optime.
2. Definirea recursivă a valorii unei soluţii optime.
3. Calculul valorii unei soluţii optime într-o manieră de tip "bottom-up".
4. Construirea unei soluţii optime din informaţia calculată.

Paşii 1-3 sunt baza unei abordări de tip programare dinamică. Pasul 4 poate fi omis dacă se doreşte doar calculul unei singure soluţii optime. In vederea realizării pasului 4, deseori se păstrează informaţie suplimentară de la execuţia pasului 3, pentru a uşura construcţia unei soluţii optimale.

6.1. Înmulţirea unui şir de matrice

Primul nostru exemplu de programare dinamică este un algoritm care rezolvă problema înmulţirii unui şir de matrice. Se dă un şir (o secvenţă) $(A_1, A_2, ..., A_n)$ de n matrice care trebuie înmulţite, şi se doreşte calcularea produsului

$$A_1 A_2 ... A_n \qquad (6.1)$$

Expresia (3.1) se poate evalua folosind algoritmul standard de înmulţire a unei perechi de matrice ca subrutină, o dată ce expresia a fost parantezată (parantezarea este necesară pentru a exclude toate ambiguităţile privind înmulţirile de matrice). Un produs de matrice este ***complet parantezat*** fie dacă este format dintr-o unică matrice, fie dacă este produsul a două produse de matrice care sunt la rândul lor complet parantezate. Cum înmulţirea matricelor este asociativă, toate parantezările conduc la acelaşi produs. De exemplu, dacă şirul de matrice este (A_1, A_2, A_3, A_4), produsul $A_1 A_2 A_3 A_4$ poate fi complet parantezat în cinci moduri distincte, astfel avem:

$(A_1(A_2(A_3 A_4)))$, $(A_1((A_2 A_3)A_4)$, $((A_1 A_2)(A_3 A_4))$, $((A_1(A_2 A_3))A_4)$, $(((A_1 A_2)A_3)A_4)$.

Să considerăm mai întâi costul înmulţirii a două matrice. Algoritmul standard este dat de următoarea procedură, descrisă în pseudocod. Prin *linii* şi *coloane* sunt referite numărul de linii, respectiv de coloane ale matricei.

ÎNMULŢIRE-MATRICE(*A*, *B*)
1. **if** *coloane*[*A*] ≠ *linii* [*B*]
2. **then** mesaj de eroare: „*dimensiuni incompatibile*"
3. **else**
4. **for** *i*←1**to** *linii*[*A*]
5. **do for** *j* ←1**to** *coloane*[*B*]
6. **do** *C*[*i*, *j*] ← 0
7. **for** *k* ←1 **to** *coloane*[*A*]
8. **do** *C*[*i*, *j*]← *C*[*i*, *j*] + *A*[*i*, *k*] * *B*[*k*, *j*]
9. **return** *C*

Două matrice *A* şi *B* se pot înmulţi numai dacă numărul de coloane din *A* este egal cu numărul de linii din *B*. Astfel, dacă *A* este o matrice având dimensiunea p x q şi *B* este o matrice având dimensiunea q x r, matricea produsă *C* este o matrice având dimensiunea p x r. Timpul necesar calculului matricei *C* este determinat de numărul de înmulţiri scalare (a se vedea linia 8 din algoritm) care este pqr. In cele ce urmează, vom exprima timpul de execuţie în funcţie de numărul de înmulţiri scalare.

Pentru a ilustra modul în care apar costuri diferite la parantezări diferite ale produsului de matrice considerăm problema şirului (A_1, A_2, A_3) de trei matrice. Presupunem că dimensiunile matricelor sunt 10 x 100, 100 x 5 şi, respectiv, 5 x 50. Dacă efectuăm înmulţire conform parantezării ((A_1A_2)A_3), atunci vom avea 10 • 100 • 5 = 5000 înmulţiri scalare pentru a calcula matricea A_1A_2 de dimensiune 10 x 5 plus alte 10 • 5 • 50 = 2500 înmulţiri scalare pentru a înmulţi această matrice cu matricea A_3. Astfel rezultă un total de 7500 înmulţiri scalare. Dacă însă vom efectua înmulţirile conform parantezării (A_1(A_2A_3)), vom avea 100 • 5 • 50 = 25.000 înmulţiri scalare pentru a calcula matricea A_2A_3 de dimensiuni 100 x 50 plus alte 10 • 100 • 50 = 50.000 înmulţiri scalare pentru a înmulţi A_1 cu această matrice. Astfel rezultă un total de 75.000 înmulţiri scalare. Deci, calculând produsul conform primei parantezări, calculul este de 10 ori mai rapid.

Problema înmulţirii şirului de matrice poate fi enunţată în următorul mod: dându-se un şir (A_1, A_2, ..., A_n) de n matrice, unde pentru i = 1,2,..., n, matricea A_i are dimensiunile p_{i-1} x p_i să se parantezeze complet produsul $A_1A_2\cdots A_n$, astfel încât să se minimizeze numărul de înmulţiri scalare.

6.1.1. Evaluarea numărului de parantezări

Înainte de a rezolva problema înmulţirii şirului de matrice prin programare dinamică, trebuie să ne convingem că verificarea exhaustivă a tuturor parantezărilor nu conduce la un algoritm eficient. Fie $P(n)$ numărul de parantezări distincte ale unei secvenţe de n matrice. Deoarece secvenţa de matrice o putem diviza între matricele k şi (k+1) pentru orice k = 1, 2, ..., n– 1 şi apoi putem descompune în paranteze, în mod independent, fiecare dintre cele două subsecvenţe, astfel obţinem recurenţa:

$$P(n) = \begin{cases} 1 \text{ dacă } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) \text{ dacă } n \geq 2. \end{cases}$$

Soluţia acestei recurenţe este secvenţa de ***numere Catalan:***

$$P(n) = C(n-1),$$

unde

$$\boldsymbol{C(n) = \frac{1}{n+1}\binom{2n}{n} = \Omega(4^n/n^{3/2})}$$

Numărul de soluţii este exponenţial în n şi metoda forţei brute a căutării exhaustive este, prin urmare, o strategie slabă de determinare a modalităţii de parantezare a şirului de matrice.

6.1.2. Structura unei parantezări optime

Primul pas din schema generală a metodei programării dinamice constă în caracterizarea structurii unei soluţii optimale. Pentru problema înmulţirii şirului de matrice, aceasta este descrisă în continuare. Pentru simplitate, vom adopta convenţia de notare $A_{i..j}$ pentru matricea care rezultă în urma evaluării produsului $A_iA_{i+1}\cdots A_j$. O parantezare optimă a produsului $A_1A_2\cdots A_n$ împarte produsul între A_k şi A_{k+1} pentru un anumit întreg k din intervalul $1 \leq k < n$. Aceasta înseamnă că, pentru o valoare a lui k, mai întâi calculăm matricele $A_{1..k}$ şi $A_{k+1..n}$ şi apoi, le înmulţim pentru a produce rezultatul final $A_{1..n}$. Costul acestei parantezări optime este costul calculului matricei $A_{1..k}$ plus costul calculului matricei $A_{k+1..n}$, plus costul înmulţirii celor două matrice.

Observaţia cheie este că parantezarea subşirului "prefix" $A_1A_2\cdots A_k$, în cadrul parantezării optime a produsului $A_1A_2\cdots A_n$, trebuie să fie o parantezare *optimă* pentru $A_1A_2\cdots A_k$ deoarece dacă ar fi existat o modalitate mai puţin costisitoare de parantezare a lui $A_1A_2\cdots A_k$, înlocuirea respectivei parantezări în parantezarea lui $A_1A_2\cdots A_n$ ar produce o altă parantezare pentru $A_1A_2\cdots A_n$ al cărei cost ar fi mai mic decât costul optimal, ceea ce este o contradicţie. O observaţie asemănătoare este valabilă şi pentru parantezarea subşirului $A_{k+1}A_{k+2}\cdots A_n$ în cadrul parantezării optime a lui $A_1A_2\cdots A_n$: aceasta trebuie să fie o parantezare optimă pentru $A_{k+1}A_{k+2}\cdots A_n$.

Prin urmare, o soluţie optimă a unei instanţe a problemei înmulţirii şirului de matrice conţine soluţii optime pentru instanţe ale subproblemelor. Existenţa substructurilor optime în cadrul unei soluţii optime este una dintre caracteristicile cadrului de aplicare a metodei programării dinamice.

6.1.3. O soluţie recursivă

Al doilea pas în aplicarea metodei programării dinamice este definirea valorii unei soluţii optime în mod recursiv, în funcţie de soluţiile optime ale subproblemelor. în cazul problemei înmulţim şirului de matrice, o subproblemă constă în determinarea costului minim al unei parantezări a şirului $A_iA_{i+1}\cdots A_j$, pentru $1 \le i \le j \le$ n. Fie $m[i, j]$ numărul minim de înmulţiri scalare necesare pentru a calcula matricea $A_{i..j}$; costul modalităţii optime de calcul a lui $A_{1..n}$ va fi, atunci, $m[1, n]$.

Putem defini $m[i, j]$ în mod recursiv, după cum urmează. Dacă $i = j$ şirul constă dintr-o singură matrice $A_{i..i} = A_i$ şi, pentru calculul produsului nu este necesară nici o înmulţire scalară. Atunci, $m[i, i] = 0$ pentru $i = 1,2, ..., n$. Pentru a calcula $m[i, j]$ când $i < j$, vom folosi structura unei soluţii optimale găsite la pasul 1. Să presupunem că descompunerea optimă în paranteze împarte produsul $A_iA_{i+1}\cdots A_j$ între A_k şi A_{k+1}, cu $i \le k < j$. Atunci $m[i, j]$ este egal cu costul minim pentru calculul subprodusului $A_{i..k}$ şi $A_{k+1..j}$, plus costul înmulţirii acestor două matrice rezultat. Deoarece evaluarea produsului $A_{i..k}A_{k+1..j}$ necesită $p_{i-1}p_kp_j$ înmulţiri scalare, vom obţine

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j.$$

Aceasta ecuaţie recursivă presupune cunoaşterea valorii lui k, lucru care nu este posibil. Există doar $j - i$ valori posibile pentru k, şi anume $k = i, i + 1,... ,j - 1$. Deoarece parantezarea optimă trebuie să folosească una dintre aceste valori pentru k, va trebui să le verificăm pe toate pentru a o putea găsi pe cea mai bună. Atunci, definiţia recursivă a costului minim a parantezării produsului $A_iA_{i+1}\cdots A_j$, devine

$$m(i,j) = \begin{cases} 0 & i = j \\ min\{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & i < j \end{cases} \qquad (6.2)$$

Valorile $m[i,j]$ exprimă costul soluţiilor optime ale subproblemelor. Pentru a putea urmări modul de construcţie a soluţiei optime, să definim $s[i,j]$ care va conţine valoarea k pentru care împărţirea produsului $A_iA_{i+1}\cdots A_j$ produce o parantezare optimă. Aceasta înseamnă că $s[i,j]$ este egal cu valoarea k pentru care $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$.

6.1.4. Calculul costului optimal

În acest moment, este uşor să scriem un algoritm recursiv bazat pe recurenţa (6.2) care calculează costul minim $m[1,n]$ al produsului $A_1A_2\cdots A_n$. După cum vom vedea mai târziu, acest algoritm necesită un timp exponenţial –nu mai bun decât metoda forţei brute folosită pentru verificarea fiecărui mod de descompunere in paranteze a produsului de matrice.

Observaţia importantă care se impune în acest moment este că avem relativ puţine subprobleme: o problemă pentru fiecare alegere a lui i şi j ce satisfac $1 \le i \le j \le$ n, adică un total de$\binom{n}{2}$+n=$\Theta(n^2)$. Un algoritm recursiv poate întâlni fiecare subproblemă de mai multe ori pe ramuri diferite ale arborelui său de recurenţă. Această proprietate de suprapunere a subproblemelor este a doua caracteristică a programării dinamice.

În loc să calculăm recursiv soluţia recurenţei (4.2), vom aplica cel de-al treilea pas al schemei generale a metodei programării dinamice şi vom calcula costul optimal cu o abordare "bottom-up". Algoritmul următor presupune că matricea A, are dimensiunile p_{i-1} x p_i, pentru $i = 1,2,...,n$. Intrarea este secvenţa $(p_0,p_1, ... ,p_n)$, unde $lungime[p] = n + 1$. Procedura foloseşte un tablou auxiliar $m[1..n, 1..n]$ pentru costurile $m[i,j]$ şi un tablou auxiliar $s[1..n, 1..n]$ care înregistrează acea valoare a lui k pentru care s-a obţinut costul optim în calculul lui $m[i,j]$.

ORDINE –ŞIR –MATRICE(p)

```
1: n ← lungime[p] -1
2: for i ← 1 to n do
3:   m[i,i]←0
4: for l ← 2 to n do
5:   for i ← 1 to n –l + 1 do
6:      j ← i +l –1
```

```
7:        m[i,j] ← ∞
8:        for k ← i to j–1  do
9:            q ← m[i, k] + m[k + 1,j] + p_{i-1}p_kp_j
10:          if q < m[i,j] then
11:              m[i,j] ← q
12:               s[i,j] ← k
13: return m, s
```

Algoritmul completează tabloul *m* într-un mod ce corespunde rezolvării problemei parantezării unor şiruri de matrice de lungime din ce în ce mai mare. Ecuaţia (6.2) arată că m*[i,j],* costul de calcul al produsului şirului de $j-i+1$ matrice, depinde doar de costurile calculării produselor şirurilor de mai puţin de $j-i+1$ matrice. Aceasta înseamnă că, pentru $k = i, i + 1,\ldots ,j - 1$, matricea $A_{i..k}$. este un produs de $k-i+1<j-i +1$ matrice, iar matricea $A_{k+i..j}$ este un produs de $j-k < j-i + 1$ matrice.

Algoritmul iniţializează, mai întâi, $m[i, i] \leftarrow 0$ pentru $i = 1, 2,\ldots, n$ (costul minim al şirurilor de lungime 1) în liniile 2-3. Se foloseşte, apoi, recurenţa (6. 2) pentru a calcula $m[i,i+ 1]$ pentru $i = 1,2,\ldots, n - 1$ (costul minim al şirurilor de lungime 2) la prima execuţie a ciclului din liniile 4 –12. La a doua trecere prin ciclu, se calculează $m[i,i+2]$ pentru $i = 1,2,\ldots,n-2$ (costul minim al şirurilor de lungime 3) etc. La fiecare pas, costul $m[i,j]$ calculat pe liniile 9 –12 depinde doar de intrările $m[i,k]$ şi $m[k + 1,j]$ ale tabloului, deja calculate.

Figura 6.1 ilustrează această procedură pe un şir de $n = 6$ matrice. Deoarece am definit $m[i,j]$ doar pentru $i \leq j$, din tabloul *m* folosim doar porţiunea situată strict deasupra diagonalei principale. Figura arată tablourile rotite astfel încât diagonala principală devine orizontală, iar şirul de matrice este înscris de-a lungul liniei de jos. Folosind acest model, costul minim $m[i,j]$ *de* înmulţire a subşirului $A_iA_{i+1}\cdots A_j$, de matrice poate fi găsit la intersecţia liniilor care pornesc

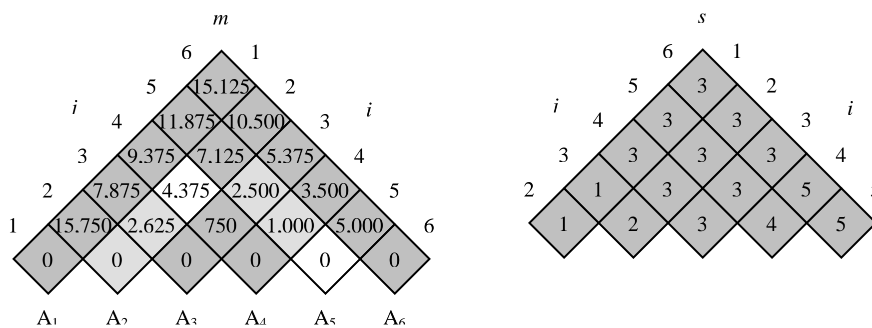


Figura 6.1 Tabelele *m* şi *s* calculate de ORDINE-ŞIR-MATRICE pentru $n = 6$ şi matricele cu dimensiunile date mai jos:

matrice	dimensiuni
A_1	30x35
A_2	35x15
A_3	15x5
A_4	5x10
A_5	10x20
A_6	20x25

Tabelele sunt rotite astfel încât diagonala principală devine orizontală. în tabloul m sunt utilizate doar elementele de deasupra diagonalei principale, iar în tabloul *s* sunt utilizate doar elementele de sub diagonala principală. Numărul minim de înmulţiri scalare necesare pentru înmulţirea a şase matrice este $m[1,6] = 15.125$. Dintre pătratele haşurate cu gri deschis, perechile care sunt colorate cu aceeaşi nuanţă sunt considerate împreună in linia 9, când calculăm pe direcţia nord-est din A_i şi pe direcţia nord-vest din *Aj*. Fiecare rând orizontal din tablou conţine valorile pentru şiruri de matrice de aceeaşi lungime. Procedura ORDINE-ŞIR-MATRICE calculează rândurile de jos în sus şi de la stânga la dreapta in cadrul fiecărui rând. 0 valoare $m[i,j]$ este calculată pe baza folosirii produselor $p_{i-1}p_kp_j$ pentru $k=i, i+1,\ldots ,j-1$ şi a tuturor valorilor aflate pe direcţia sud-est şi sud-vest faţă de $m[i,j]$.

$$m[2,5] = min \begin{cases} m[2,2] + m[3,5] + p_1p_2p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13000 \\ m[2,3] + m[4,5] + p_1p_3p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125 \\ m[2,4] + m[5,5] + p_1p_4p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11375 \end{cases} = 7125$$

O simplă examinare a structurii de ciclu imbricat din ORDINE-ŞIR-MATRICE conduce la constatarea că timpul de execuţie a algoritmului este $O(n^3)$. în fiecare dintre cele trei cicluri imbricate, indicii acestora *(l, i* şi *k)* iau cel mult *n* valori.

6.1.5. Construirea unei soluţii optime

Deşi algoritmul ORDINE-ŞIR-MATRICE determină numărul optim de înmulţiri se necesare pentru calculul produsului şirului de matrice, acesta nu prezintă în mod direct, trebuie făcută înmulţirea. Pasul 4 al schemei generale a metodei programării dinamice urmează construirea unei soluţii optime din informaţia disponibilă.

În acest caz particular, vom folosi tabloul *s[1..n,1..n]* pentru a determina modul optim de înmulţire a matricelor. Fiecare element *s[i,j]* conţine valoarea lui *k* pentru care parantezarea optimă a produsului $A_iA_{i+1}\cdots A_j$ împarte produsul între A_k şi A_{k+1}. Atunci ştim că, în produsul final de calcul al matricei $A_{1..n}$, optimul este $A_{1..s[1,n]}A_{s[1,n]+1..n}$. Înmulţirile anterioare determinate recursiv, deoarece *s[1,s[1,n]]* determină ultima înmulţire matriceală din calculul lui $A_{1..s[1,n]}$ şi *s[s[1,n]+1,n]* determină ultima înmulţire matriceală din calculul lui $A_{s[1,n]}$ - .Următoarea procedură recursivă calculează produsul şirului de matrice $A_{i..j}$, fiind date matricea $A = (A_1, A_2, ..., A_n)$, tabloul *s* calculat de ORDINE-ŞIR-MATRICE Şi indicii *i* şi *j*. Apelul este ÎNMULŢIRE-ŞIR-MATRICE (*A*,*s*,1,*n*).

ÎNMULŢIRE-ŞIR-MATRICE (*A*,*s*,1,*n*)
1: **if** *j* > *i* **then**
2: X ← ÎNMULŢIRE-ŞIR-MATRICE (*A*, *s*, *i*, *s[i,j]*)
3: *Y* ← ÎNMULŢIRE-ŞIR-MATRICE (*A*, *s*, *s[i,j]*+1, *j*)
4: **return** ÎNMULŢIRE-ŞIR-MATRICE (*X*,*Y*)
5: **else**
6: **return** A_i

În exemplul din figura 6.1, apelul ÎNMULŢIRE-ŞIR-MATRICE (*A*,*s*,1,6) calculează produsul şirului de matrice conform descompunerii în paranteze $((A_1(A_2A_3))((A_4A_5)A_6))$.

6.2. Elemente de programare dinamică

Cu toate că am prezentat un exemplu de aplicare a metodei programării dinamice, este posibil să existe, în continuare, întrebări asupra situaţiilor în care ea este aplicabilă, sau, altfel spus, când trebuie căutată o soluţie de programare dinamică pentru o anumită problemă? În această secţiune se vor examina două componente de bază pe care trebuie să le prezinte problema de optimizare pentru ca programarea dinamică să fie aplicabilă: substructura optimă şi subproblemele suprapuse. Se va urmări, însă, şi o variantă a metodei, numită memoizare, în care se utilizează proprietatea de suprapunere a subproblemelor.

Substructura optimă

Primul pas al rezolvării unei probleme de optimizare prin programare dinamică constă în caracterizarea structurii unei soluţii optime. Spunem că problema prezintă o ***substructură optimă*** dacă o soluţie optimă a problemei include soluţii optime ale subproblemelor. Ori de câte ori o problemă prezintă o structură optimă, acesta este un bun indiciu pentru posibilitatea aplicării programării dinamice.

În secţiunea 6.1 am văzut că problema înmulţirii şirului de matrice prezintă o substructură optimă. Am observat că o parantezare optimă pentru $A_1A_2\cdots A_n$, care separă produsul între A_k şi A_{k+1}, include soluţii optime pentru problemele parantezării matricelor $A_1A_2\cdots A_k$ şi $A_{k+1}A_{k+2}\cdots A_n$. Tehnica utilizată pentru a arăta că subproblemele au soluţii optime este clasică. Presupunem că există o soluţie mai bună pentru o subproblemă şi arătăm că această presupunere contrazice optimalitatea soluţiei problemei iniţiale.

Substructura optimă a unei probleme sugerează, deseori, un spaţiu potrivit al subproblemelor pentru care se poate aplica programarea dinamica. De obicei, există mai multe clase de subprobleme care pot fi considerate "naturale" pentru o problemă. De exemplu, spaţiul subproblemelor pe care le-am considerat la înmulţirea şirului de matrice conţinea toate subsecvenţele şirului de intrare. La fel de bine, am fi putut alege ca spaţiu al subproblemelor secvenţe arbitrare de matrice din şirul de intrare, dar acest spaţiu al subproblemelor ar fi fost prea mare. Un algoritm de programare dinamică bazat pe acest spaţiu al subproblemelor ar rezolva mult mai multe probleme decât ar fi necesar.

Investigarea substructurii optime a unei probleme prin iterarea instanţelor subproblemelor acesteia este o metodă bună pentru a găsi un spaţiu potrivit de subprobleme pentru programarea dinamică. De exemplu, după examinarea structurii unei soluţii optime a problemei înmulţirii şirului de matrice, putem itera şi examina structura soluţiilor optime ale subproblemelor, a subsubproblemelor şi aşa mai departe. Descoperim, astfel, că toate subproblemele sunt compuse din subsecvenţe ale şirului $(A_1A_2\cdots A_n)$. Deci, mulţimea şirurilor de forma $(A_iA_{i+1}\cdots A_j)$ cu $1 \le i \le j \le n$ un spaţiu de subprobleme natural şi rezonabil de utilizat.

6.2.1. Suprapunerea subproblemelor

O a doua componentă pe care trebuie să o aibă o problemă de optimizare pentru ca programarea dinamică să fie aplicabilă este ca spaţiul subproblemelor să fie "restrâns", în sensul că un algoritm recursiv rezolvă mereu aceleaşi subprobleme, în loc să genereze subprobleme noi.

De obicei, numărul total de subprobleme distincte este dependent polinomial de dimensiunea datelor de intrare. Când un algoritm recursiv abordează mereu o aceeaşi problemă, se spune că problema de optimizare are ***subprobleme suprapuse.*** Spre deosebire de aceasta, o problemă care se rezolvă cu un algoritm de tip divide şi stăpâneşte generează, la fiecare nouă etapă. probleme noi. În general, algoritmii de programare dinamică folosesc suprapunerea subproblemelor prin rezolvarea o singură dată a fiecărei subprobleme, urmată de stocarea soluţiei într-un tablou unde poate fi regăsită la nevoie, necesitând pentru regăsire un timp constant.

Pentru a ilustra proprietatea de subprobleme suprapuse, vom reexamina problema înmulţirii şirului de matrice. Revenind la figura 4.l. observăm că algoritmul ORDINE-ŞIR-MATRICE preia, în mod repetat, soluţiile subproblemelor din liniile de jos atunci când rezolvă subproblemele din liniile superioare. De exemplu, valoarea m[3,4] este referită de patru ori: în timpul calculelor pentru *m*[2,4], *m*[1,4], *m*[3,5] şi *m*[3,6]. Dacă *m*[3,4] ar fi recalculată de fiecare dată in loc să fie preluată din tablou, creşterea timpului de execuţie ar fi dramatică.

6.3. Cel mai lung subşir comun

Următoarea problemă pe care o vom lua în considerare este problema celui mai lung subşir comun. Un subşir al unui şir dat este şirul iniţial din care lipsesc unele elemente (eventual nici unul). În mod formal, dându-se şirul $X = (x_1x_2,\ldots ,X_m)$, un alt şir $Z = (Z_1,Z_2,\ldots ,Z_k)$ este un *subşir* al lui *X* dacă există un şir strict crescător de indici din *X*, $(i_1,i_2,\ldots,i_k)$, astfel încât, pentru toate valorile $j= 1, 2,\ldots, k$, avem $x_{i_j} = z_j$. De exemplu, $Z = (B, C, D, B)$ este un subşir al lui $X = (A, B, C, B, D, A, B)$, secvenţa de indici corespunzătoare fiind (2,3,5,7).

Dându-se două şiruri *X* şi *Y*, spunem că şirul *Z* este un *subşir comun* pentru *X şi Y* dacă *Z* este un subşir atât pentru *X* cât şi pentru *Y*. De exemplu, dacă $X = (A, B, C, B, D, A, B)$ şi $Y = (B, D, C, A, B, A)$, şirul (B, C, A) este un subşir comun pentru *X* şi *Y*. Şirul (B, C, A) nu este *cel mai lung* subşir comun (CMLSC) pentru *X* şi *Y*, deoarece are lungimea 3, iar şirul (B, C, B, A), care este de asemenea subşir comun lui *X* şi *Y*, are lungimea 4. Şirul (B, C, B, A) este un CMLSC pentru *X* şi *Y*, la fel şi şirul (B, D, A, B), deoarece nu există un subşir comun de lungime cel puţin 5.

În problema *celui mai lung subşir comun,* se dau două şiruri $X = (x_1x_2,\ldots ,x_m)$ şi $Y = (y_1,y_2,\ldots,y_n)$ şi se doreşte determinarea unui subşir comun de lungime maximă pentru *X* şi *Y*. Această secţiune va arăta că problema CMLSC poate fi rezolvată eficient prin metoda programării dinamice.

6.3.1. Caracterizarea celui mai lung subşir comun

O abordare prin metoda forţei brute pentru rezolvarea problemei CMLSC constă în enumerarea tuturor subşirurilor lui *X* şi verificarea dacă acestea constituie un subşir şi pentru *Y*, memorând cel mai lung subşir găsit. Fiecare subşir al lui *X* corespunde unei submulţimi a indicilor lui *X* $(1,2,\ldots, m)$. Există 2^m subşiruri pentru *X* şi, deci, această abordare necesită un timp exponenţial, ceea ce o face inabordabilă pentru şiruri de lungimi mari.

Problema CMLSC are proprietatea de substructură optimă, aşa cum vom arăta în următoarea teoremă. Clasa naturală de subprobleme corespunde unor perechi de "prefixe" ale celor două şiruri de intrare. Mai precis, dându-se un şir $X = (x_1x_2,\ldots ,x_m)$, definim *prefixul i* al lui *X*, pentru $i = 0,1,\ldots, m$, ca fiind $X_i = (x_1x_2,\ldots ,x_i)$. De exemplu, dacă $X=(A, B, C, B, D, A, B)$, atunci $X_4 =(A, B, C, B)$ iar X_0 este şirul vid.

Teorema 6.1 (Substructura optimală a unui CMLSC) Fie şirurile $X = (x_1x_2,\ldots ,x_m)$ şi $Y = (y_1,y_2,\ldots,y_n)$ şi fie $Z=(Z_1,Z_2,\ldots ,Z_k)$ un CMLSC pentru *X* şi *Y*.

1. Dacă $x_m \neq y_n$, atunci $z_k = x_m = y_n$ şi Z_{k-1} este un CMLSC pentru X_{m-1} şi Y_{n-1}.
2. Dacă $x_m \neq y_n$, atunci, din $z_k \neq x_m$ rezultă că *Z* este un CMLSC pentru X_{m-1} şi *Y*.
3. Dacă $x_m \neq y_n$, atunci, din $z_k \neq y_n$ rezultă că *Z* este un CMLSC pentru *X* şi Y_{n-1}.

Demonstraţie:

(1) Dacă $z_k \neq x_m$ atunci putem adăuga $x_m = y_n$ la *Z* pentru a obţine un subşir comun a lui *X* şi *Y* de lungime k + 1. contrazicând presupunerea că *Z* este *cel mai lung* subşir comun pentru *X* şi *Y*. Deci, va trebui să avem $z_k = x_m = y_n$. Prefixul Z_{k-1} este un subşir comun de lungime $k-1$ pentru X_{m-1} şi Y_{n-1}. Dorim să demonstrăm că este un CMLSC. Să presupunem, prin absurd. că există un subşir *W* comun pentru X_{m-1} şi Y_{n-1}, a cărui lungime este fie mai mare decât $k-1$. Atunci, adăugând la *W* elementul $x_m = y_n$ se va forma un subşir comun a lui *X* şi *Y* a cărui lungime este mai mare decât *k*, ceea ce este o contradicţie.

(2) Dacă $z_k \neq x_m$, atunci *Z* este un subşir comun pentru X_{m-1} şi *Y* . Dacă ar exista un subşir comun *W* al lui X_{m-1} şi *Y* , cu lungime mai mare decât *k,* atunci *W* ar fi şi un subşir comun al lui X_m şi *Y,* contrazicând presupunerea că *Z* este un CMLSC pentru *X* şi *Y*.

(3) Demonstraţia este simetrică celei de la (2).

Caracterizarea din teorema 6.1 arată că un CMLSC al două şiruri conţine un CMLSC pentru prefixele celor două şiruri. Atunci, problema CMLSC are proprietatea de substructură optimală. O soluţie recursivă are, de asemenea, proprietatea de suprapunere a problemelor, după cum vom arăta în cele ce urmează.

6.3.2. O soluţie recursivă a subproblemelor

Teorema 6.1 implică faptul că există fie una, fie două probleme care trebuie examinate pentru găsirea unui CMLSC pentru $X = (x_1 x_2, \ldots, x_m)$ şi $Y = (y_1, y_2, \ldots, y_n)$. Dacă $x_m = y_n$, trebuie găsit un CMLSC pentru X_{m-1} şi Y_{n-1}. Adăugarea elementului $x_m = y_n$ la acest CMLSC produce un CMLSC pentru X şi Y. Dacă $x_m \neq y_n$, atunci trebuie rezolvate două subprobleme: găsirea unui CMLSC pentru X_{m-1} şi Y şi găsirea unui CMLSC pentru X şi Y_{n-1}. Cel mai lung CMLSC dintre acestea două va fi CMLSC pentru X şi Y.

Se poate observa că problema CMLSC se descompune în subprobleme suprapuse. Pentru a găsi un CMLSC al şirurilor X şi Y, va trebui, probabil, calculat CMLSC pentru X şi i Y_{n-1} respectiv, pentru X_{m-1} şi Y . Dar fiecare dintre aceste subprobleme conţine sub-subproblema găsirii CMLSC pentru X_{m-1} şi Y_{n-1}. Multe alte subprobleme au în comun sub-subprobleme.

Ca şi problema înmulţirii şirului de matrice, soluţia recursivă pentru problema CMLSC implică stabilirea unei recurenţe pentru costul unei soluţii optime. Să definim *c[i,j]* ca lungimea unui CMLSC al şirurilor X_i, şi Y_j. Dacă i=0 sau j=0, CMLSC are lungimea 0. Substructura optimală a problemei CMLSC produce formula recursivă

$$c[i,j] = \begin{cases} 0, & i = 0,\ j = 0, \\ c[i-1, j-1] + 1, & i,j > 0 \ and \ x_i = y_j, \\ max(c[i-1], c[i-1,j]), & i,j > 0 \ and \ x_i = y_j. \end{cases} \tag{6.2}$$

6.3.3. Calculul lungimii unui CMLSC

Pe baza ecuaţiei (4.5) se poate scrie un algoritm recursiv având timp exponenţial pentru calculul lungimii unui CMLSC a două şiruri. Deoarece există numai $\Theta(mn)$ subprobleme distincte, vom folosi metoda programării dinamice pentru a calcula soluţiile în mod "bottom-up".

LUNGIME-CMLSC(A', R)
1: $m \leftarrow$ *lungime[X]*
2: $n \leftarrow$ *lungime[Y]*
3: **pentru** $i \leftarrow 1, m$ **execută**
4: *c[i,0]* ← 0
5: **pentru** $j \leftarrow 0$, n **execută**
6: *c[0,j]* ← 0
7: **pentru** $i \leftarrow 1$, m **execută**
8: **pentru** $j \leftarrow 1$, n **execută**
9: **dacă** $x_i = y_j$ **atunci**
10: *c[i,j]* ← *c[i –1,j –1]*+ 1
11: *b[i, j]*← „↖"
12: **altfel**
13: **dacă** *c[i -1,j]* ≥ *c[i, j –1]* **atunci**
14: *c[i, j]* ← *c[i-1, j]*
15: *b[i, j]* ← „↑"
16: **altfel**
17: *c[i, j]* ← *c[i, j –1]*
18: *b[i, j]* ← „←"
19: **returnează** c, *b*

Procedura LUNGIME-CMLSC are ca date de intrare două şiruri $X = (x_1 x_2, \ldots, x_m)$ şi $Y = (y_1, y_2, \ldots, y_n)$. Procedura memorează valorile *c[i, j]* într-un tablou *c[0..m,0..n]* ale cărui elemente sunt calculate în ordinea crescătoare a liniilor. (Aceasta înseamnă că se completează mai întâi prima linie de la stânga la dreapta, apoi a doua linie şi aşa mai departe.) De asemenea. se construieşte un tablou *b[1..m, 1..n]* care simplifică determinarea unei soluţii optimale. Intuitiv. *b[i, j]* indică elementul tabloului care corespunde soluţiei optime a subproblemei alese la calculul lui *c[i, j]*. Procedura returnează tablourile *b* şi *c*; *c[m, n]* conţine lungimea unui CMLSC pentru *X* şi *Y*.

Figura 16.3 conţine tabloul produs de LUNGIME-CMLSC pentru şirurile *X* = (*A*, *B*, *C*, *B*, *D*, *A*, *B)* şi *Y* = (*B*, *D*, *C*, *A*, *B*, *A*). Timpul de execuţie al procedurii este *O(mn)*, deoarece calculul fiecărui element al tabloului necesită un timp *O(1)*.

Construirea unui CMLSC

Tabelul b returnat de LUNGIME-CMLSC poate fi folosit pentru construcţia rapidă a unui CMLSC pentru $X = (x_1 x_2,\ldots ,x_m)$ şi $Y = (y_1,y_2,\ldots,y_n)$. Pur şi simplu, se începe cu $b[m, n]$ şi se parcurge tabloul conform direcţiilor indicate. Ori de câte ori se întâlneşte un element "↖" pe poziţia $b[i, j]$, aceasta înseamnă că $x_i = y_j$ este un element al CMLSC. Prin această metodă, elementele CMLSC sunt regăsite în ordine inversă. Următoarea procedură recursivă tipăreşte un CMLSC al şirurilor X şi Y în ordinea corectă. Apelul iniţial este SCRIE-CMLSC*(b, X, lungime[X], lungime[Y]).*

SCRIE-CMLSC (6,A,i,J)
l: **dacă** $i = 0$ sau $j = 0$ **atunci**
2: revenire
3: **dacă** $b[i, j]$ ="↖" **atunci**
4: SCRIE-CMLSC (b, X, i –1, j –1)
5: tipăreşte x_i
6: **altfel dacă** $b[i, j]$ ="↑" **atunci**
7: SCRIE-CMLSC (b, X, i –1, j)
8: altfel
9: SCRIE-CMLSC (b, X, i, j –1)

Pentru tabloul b din figura 16.3, această procedură tipăreşte *"BCBA"*. Procedura necesită un timp de $O(m + n)$, deoarece, cel puţin unul dintre i şi j este decrementat în fiecare etapa a recurenţei.

Îmbunătăţirea codului

0 dată ce s-a construit un algoritm, se descoperă că, deseori, timpul de execuţie sau spaţiul folosit pot fi îmbunătăţite. Acest lucru este adevărat, mai ales. pentru programarea dinamică clasică. Anumite modificări pot simplifica implementarea şi pot îmbunătăţi factorii constanţi, dar nu pot aduce o îmbunătăţire asimptotică a performanţei. Alte modificări pot aduce îmbunătăţiri asimptotice substanţiale atât pentru timpul de execuţie cât şi pentru spaţiul ocupat. De exemplu, se poate elimina complet tabloul b. Fiecare element $c[i, j]$ depinde de numai trei alte elemente ale tabloului c: $c[i –1, j –1]$, $c[i –1, j]$ şi $c[i, j –1]$. Dându-se valoarea lui $c[i, j]$, se poate determina, în timp $O(1)$, care dintre aceste trei valori a fost folosită pentru calculul lui $c[i, j]$ fără a mai folosi tabloul b.

j	0	1	2	3	4	5	6
	y_j	*B*	*D*	*C*	*A*	*B*	*A*
x_i			0	0	0	0	0
A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Figura 4.3 Tabelele c şi b calculate de LUNGIME-CMLSC pentru şirurile $X = (A, B, C, B, D, A, B)$ şi $Y = (B, D, C, A, B, A)$. Pătratul din linia i şi coloana j conţine valoarea $c[i, j]$ precum şi săgeata potrivită pentru valoarea lui $b[i,j]$. Valoarea 4 a lui $c[7, 6]$ - colţul cel mai din dreapta jos în tablou-este lungimea unui CMLSC (B,C,B, A) al lui X şi Y. Pentru $i, j > 0$, valoarea $c[i, j]$ depinde doar de valoarea expresiei $x_i = y_j$ şi de valorile elementelor $c[i –1,j]$, $c[i,j –1]$ şi $c[i –1,j –1]$, care sunt calculate înaintea lui $c[i, j]$. Pentru a reconstrui elementele unui CMLSC, se urmăresc săgeţile corespunzătoare lui $b[i, j]$ începând cu colţul cel mai din dreapta; drumul este haşurat. Fiecare "↖" pe acest drum corespunde unui element (care este evidenţiat) pentru care $x_i = y_j$ aparţine unui CMLSC.

Ca o consecinţă, putem reconstrui un CMLSC în timp $O(m + n)$ folosind o procedură similara cu SCRIE-CMLSC. Deşi prin această metodă se eliberează un spaţiu $\Theta(mn)$, spaţiul auxiliar necesar calculului unui CMLSC nu descreşte asimptotic, deoarece este oricum necesar un spaţiu $\Theta(mn)$ pentru tabloul c.

Totuşi, spaţiul necesar pentru LUNGIME-CMLSC îl putem reduce deoarece, la un moment dat, sunt necesare doar două linii ale tabloului c: linia care este calculată şi linia precedentă. (De fapt, putem folosi doar puţin mai mult spaţiu decât cel necesar unei linii a lui c, pentru a calcula lungimea unui CMLSC) Această îmbunătăţire

funcţionează doar dacă este necesară numai lungimea unui CMLSC; dacă este nevoie şi de reconstruirea elementelor unui CMLSC, tabloul mai mic nu conţine destulă informaţie pentru a reface etapele în timp $O(m+n)$.