ALGORITMI FUNDAMENTALI

O PERSPECTIVĂ C++

RĂZVAN ANDONIE ILIE GÂRBACEA

ALGORITMI FUNDAMENTALI

O PERSPECTIVĂ C++

Editura Libris
Cluj-Napoca, 1995

Referent: **Leon Livovschi**

Coperta: **Zoltán Albert**

ISBN 973-96494-5-9

Cuvânt înainte

Evoluția rapidă și spectaculoasă a informaticii în ultimile decenii se reflectă atât în apariția a numeroase limbaje de programare, cât și în metodele de elaborare și redactare a unor algoritmi performanți.

Un concept nou, care s-a dovedit foarte eficient, este cel al programării orientate pe obiect, prin obiect înțelegându-se o entitate ce cuprinde atât datele, cât și procedurile ce operează cu ele. Dintre limbajele orientate pe obiect, limbajul C++ prezintă – printre multe altele – avantajul unei exprimări concise, fapt ce ușurează transcrierea în acest limbaj a algoritmilor redactați în pseudo-cod și motivează folosirea lui în cartea de față. Cu toate că nu este descris în detaliu, este demn de menționat faptul că descrierea din Capitolul 2, împreună cu completările din celelalte capitole, constituie o prezentare aproape integrală a limbajului C++.

O preocupare meritorie a acestei lucrări este problema analizei eficienței algoritmilor. Prezentarea acestei probleme începe în Capitolul 1 și continuă în Capitolul 5. Tehnicile de analiză expuse se bazează pe diferite metode, prezentate într-un mod riguros și accesibil. Subliniem contribuția autorilor în expunerea detailată a inducției constructive și a tehnicilor de rezolvare a recurențelor liniare.

Diferitele metode clasice de elaborare a algoritmilor sunt descrise în Capitolele 6–8 prin probleme ce ilustrează foarte clar ideile de bază și detaliile metodelor expuse. Pentru majoritatea problemelor tratate, este analizată și eficiența algoritmului folosit. Capitolul 9 este consacrat tehnicilor de explorări în grafuri. În primele secțiuni sunt prezentate diferite probleme privind parcurgerea grafurilor. Partea finală a capitolului este dedicată jocurilor și cuprinde algoritmi ce reprezintă – de fapt – soluții ale unor probleme de inteligență artificială.

Cartea este redactată clar și riguros, tratând o arie largă de probleme din domeniul elaborării și analizei algoritmilor. Exercițiile din încheierea fiecărui capitol sunt foarte bine alese, multe din ele fiind însoțite de soluții. De asemenea, merită menționate referirile interesante la istoria algoritmilor și a gândirii algoritmice.

Considerăm că această carte va fi apreciată și căutată de către toți cei ce lucrează în domeniul abordat și doresc să-l cunoască mai bine.

Leon Livovschi

În clipa când exprimăm un lucru, reușim,
în mod bizar, să-l și depreciem.

Maeterlinck

Prefață

Cartea noastră își propune în primul rând să fie un curs și nu o "enciclopedie" de algoritmi. Pornind de la structurile de date cele mai uzuale și de la analiza eficienței algoritmilor, cartea se concentrează pe principiile fundamentale de elaborare a algoritmilor: greedy, divide et impera, programare dinamică, backtracking. Interesul nostru pentru inteligența artificială a făcut ca penultimul capitol să fie, de fapt, o introducere – din punct de vedere al algoritmilor – în acest domeniu.

Majoritatea algoritmilor selectați au o conotație estetică. Efortul necesar pentru înțelegerea elementelor mai subtile este uneori considerabil. Ce este însă un algoritm "estetic"? Putem răspunde foarte simplu: un algoritm este estetic dacă exprimă mult în cuvinte puține. Un algoritm estetic este oare în mod necesar și eficient? Cartea răspunde și acestor întrebări.

În al doilea rând, cartea prezintă mecanismele interne esențiale ale limbajului C++ (moșteniri, legături dinamice, clase parametrice, excepții) și tratează implementarea algoritmilor în conceptul programării orientate pe obiect. Totuși, această carte nu este un curs complet de C++.

Algoritmii nu sunt pur și simplu "transcriși" din pseudo-cod în limbajul C++, ci sunt regândiți din punct de vedere al programării orientate pe obiect. Sperăm că, după citirea cărții, veți dezvolta aplicații de programare orientată pe obiect și veți elabora implementări ale altor structuri de date. Programele* au fost scrise pentru limbajul C++ descris de Ellis și Stroustrup în *"The Annotated C++ Reference Manual"*. Acest limbaj se caracterizează, în principal, prin introducerea claselor parametrice și a unui mecanism de tratare a excepțiilor foarte avansat, facilități deosebit de importante pentru dezvoltarea de biblioteci C++. Compilatoarele GNU C++ 2.5.8 (UNIX/Linux) și Borland C++ 3.1 (DOS) suportă destul de bine clasele parametrice. Pentru tratarea excepțiilor se pot utiliza compilatoarele Borland C++ 4.0 și, în viitorul apropiat, GNU C++ 2.7.1.

Fără a face concesii rigorii matematice, prezentarea este intuitivă, cu numeroase exemple. Am evitat, pe cât posibil, situația în care o carte de informatică începe –

* Fișierele sursă ale tuturor exemplelor – aproximativ 3400 de linii în 50 de fișiere – pot fi obținute pe o dischetă MS-DOS, printr-o comandă adresată editurii.

spre disperarea ne-matematicienilor – cu celebrul “Fie ... ”, sau cu o definiție. Am încercat, pe de altă parte, să evităm situația când totul “este evident”, sau “se poate demonstra”. Fiecare capitol este conceput fluid, ca o mică poveste, cu puține referințe și note. Multe rezultate mai tehnice sunt obținute ca exerciții. Algoritmii sunt prezentați într-un limbaj pseudo-cod compact, fără detalii inutile. Am adăugat la sfârșitul fiecărui capitol numeroase exerciții, multe din ele cu soluții.

Presupunem că cititorul are la bază cel puțin un curs introductiv în programare, nefiindu-i străini termeni precum algoritm, recursivitate, funcție, procedură și pseudo-cod. Există mai multe modalități de parcurgere a cărții. În funcție de interesul și pregătirea cititorului, acesta poate alege oricare din părțile referitoare la elaborarea, analiza, sau implementarea algoritmilor. Cu excepția părților de analiză a eficienței algoritmilor (unde sunt necesare elemente de matematici superioare), cartea poate fi parcursă și de către un elev de liceu. Pentru parcurgerea secțiunilor de implementare, este recomandabilă cunoașterea limbajului C.

Cartea noastră se bazează pe cursurile pe care le ținem, începând cu 1991, la Secția de electronică și calculatoare a Universității Transilvania din Brașov. S-a dovedit utilă și experiența noastră de peste zece ani în dezvoltarea produselor software. Colectivul de procesare a imaginilor din ITC Braşov a fost un excelent mediu în care am putut să ne dezvoltăm profesional. Le mulțumim pentru aceasta celor care au făcut parte, alături de noi, din acest grup: Sorin Cismaș, Ștefan Jozsa, Eugen Carai. Nu putem să nu ne amintim cu nostalgie de compilatorul C al firmei DEC (pentru minicalculatoarele din seria PDP-11) pe care l-am “descoperit” împreună, cu zece ani în urmă.

Ca de obicei în astfel de situații, numărul celor care au contribuit într-un fel sau altul la realizarea acestei cărți este foarte mare, cuprinzând profesorii noștri, colegii de catedră, studenții pe care am “testat” cursurile, prietenii. Le mulțumim tuturor. De asemenea, apreciem răbdarea celor care ne-au suportat în cei peste doi ani de elaborare a cărții.

Sperăm să citiți această carte cu aceeași plăcere cu care ea a fost scrisă.

Brașov, ianuarie 1995

Răzvan Andonie

Ilie Gârbacea[*]

[*] Autorii pot fi contactați prin poștă, la adresa: Universitatea Transilvania, Catedra de electronică și calculatoare, Politehnicii 1-3, 2200 Brașov, sau prin E-mail, la adresa: algoritmi&c++@lbvi.sfos.ro

Cuprins

Lista de notații

$\#T$	numărul de elemente din tabloul (sau mulțimea) T
$i \,..\, j$	interval de întregi: $\{k \in \mathbf{N} \mid i \leq k \leq j\}$
m **mod** n	restul împărțirii întregi a lui m la n
div	împărțirea întreagă
$\mid x \mid$	mărimea cazului x
log	logaritm într-o bază oarecare (în contextul notației asimptotice)
lg	logaritm în baza 2
ln	logaritm natural
$\lg^*$	logaritm iterat (vezi Secțiunea 3.5)
$\binom{n}{k}$	combinări de n luate câte k
$\mathbf{R}^*$	mulțimea numerelor reale nenegative
$\mathbf{N}^+$, $\mathbf{R}^+$	mulțimea numerelor naturale (reale) strict pozitive
B	mulțimea constantelor booleene $\{true, false\}$
$(\exists x) \mid [P(x)]$	există un x astfel încât $P(x)$
$(\forall x) \mid [P(x)]$	pentru orice x astfel încât $P(x)$
$\lfloor x \rfloor$	cel mai mare întreg mai mic sau egal cu x
$\lceil x \rceil$	cel mai mic întreg mai mare sau egal cu x
O, Θ, Ω	notație asimptotică (vezi Secțiunea 5.1.1)
$\leftarrow$	atribuire
(a, b)	muchia unui graf orientat
$\{a, b\}$	muchia unui graf neorientat

1. Preliminarii

1.1 Ce este un algoritm?

Abu Ja`far Mohammed ibn Musâ al-Khowârizmî (autor persan, sec. VIII-IX), a scris o carte de matematică cunoscută în traducere latină ca “Algorithmi de numero indorum”, iar apoi ca “Liber algorithmi”, unde “algorithm” provine de la “al-Khowârizmî”, ceea ce literal înseamnă “din orașul Khowârizm”. În prezent, acest oraș se numește Khiva și se află în Uzbechistan. Atât al-Khowârizmî, cât și alți matematicieni din Evul Mediu, înțelegeau prin *algoritm* o regulă pe baza căreia se efectuau calcule aritmetice. Astfel, în timpul lui Adam Riese (sec. XVI), algoritmii foloseau la: dublări, înjumătățiri, înmulțiri de numere. Alți algoritmi apar în lucrările lui Stifer (“Arithmetica integra”, Nürnberg, 1544) și Cardano (“Ars magna sive de reguli algebraicis”, Nürnberg, 1545). Chiar și Leibniz vorbește de “algoritmi de înmulțire”. Termenul a rămas totuși multă vreme cu o întrebuințare destul de restrânsă, chiar și în domeniul matematicii.

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

Este surprinzătoare transformarea gândirii algoritmice, dintr-un instrument matematic particular, într-o modalitate fundamentală de abordare a problemelor în domenii care aparent nu au nimic comun cu matematica. Această universalitate a gândirii algoritmice este rezultatul conexiunii dintre algoritm și calculator. Astăzi, înțelegem prin *algoritm* o metodă generală de rezolvare a unui anumit tip de problemă, metodă care se poate implementa pe calculator. În acest context, un algoritm este esența absolută a unei rutine.

Cel mai faimos algoritm este desigur algoritmul lui Euclid pentru aflarea celui mai mare divizor comun a două numere întregi. Alte exemple de algoritmi sunt metodele învățate în școală pentru a înmulți/împărți două numere. Ceea ce dă însă generalitate noțiunii de algoritm este faptul că el poate opera nu numai cu numere. Există astfel algoritmi algebrici și algoritmi logici. Până și o rețetă culinară este în esență un algoritm. Practic, s-a constatat că nu există nici un domeniu, oricât ar părea el de imprecis și de fluctuant, în care să nu putem descoperi sectoare funcționând algoritmic.

Un algoritm este compus dintr-o mulțime finită de pași, fiecare necesitând una sau mai multe operații. Pentru a fi implementabile pe calculator, aceste operații trebuie să fie în primul rând *definite*, adică să fie foarte clar ce anume trebuie executat. În al doilea rând, operațiile trebuie să fie *efective*, ceea ce înseamnă că – în principiu, cel puțin – o persoană dotată cu creion și hârtie trebuie să poată efectua orice pas într-un timp finit. De exemplu, aritmetica cu numere întregi este efectivă. Aritmetica cu numere reale nu este însă efectivă, deoarece unele numere sunt exprimabile prin secvențe infinite. Vom considera că un algoritm trebuie să se termine după un număr finit de operații, într-un timp rezonabil de lung.

Programul este exprimarea unui algoritm într-un limbaj de programare. Este bine ca înainte de a învăța concepte generale, să fi acumulat deja o anumită experiență practică în domeniul respectiv. Presupunând că ați scris deja programe într-un limbaj de nivel înalt, probabil că ați avut uneori dificultăți în a formula soluția pentru o problemă. Alteori, poate că nu ați putut decide care dintre algoritmii care rezolvau aceeași problemă este mai bun. Această carte vă va învăța cum să evitați aceste situații nedorite.

Studiul algoritmilor cuprinde mai multe aspecte:

i) *Elaborarea algoritmilor*. Actul de creare a unui algoritm este o artă care nu va putea fi niciodată pe deplin automatizată. Este în fond vorba de mecanismul universal al creativității umane, care produce noul printr-o sinteză extrem de complexă de tipul:

tehnici de elaborare (reguli) + creativitate (intuiție) = soluție.

Un obiectiv major al acestei cărți este de a prezenta diverse tehnici fundamentale de elaborare a algoritmilor. Utilizând aceste tehnici, acumulând și o anumită experiență, veți fi capabili să concepeți algoritmi eficienți.

ii) *Exprimarea algoritmilor*. Forma pe care o ia un algoritm într-un program trebuie să fie clară și concisă, ceea ce implică utilizarea unui anumit stil de programare. Acest stil nu este în mod obligatoriu legat de un anumit limbaj de programare, ci, mai curând, de tipul limbajului și de modul de abordare. Astfel, începând cu anii ‘80, standardul unanim acceptat este cel de programare structurată. În prezent, se impune standardul programării orientate pe obiect.

iii) *Validarea algoritmilor*. Un algoritm, după elaborare, nu trebuie în mod necesar să fie programat pentru a demonstra că funcționează corect în orice situație. El poate fi scris inițial într-o formă precisă oarecare. În această formă, algoritmul va fi *validat*, pentru a ne asigura că algoritmul este corect, independent de limbajul în care va fi apoi programat.

iv) *Analiza algoritmilor*. Pentru a putea decide care dintre algoritmii ce rezolvă aceeași problemă este mai bun, este nevoie să definim un criteriu de apreciere a valorii unui algoritm. În general, acest criteriu se referă la timpul de calcul și la memoria necesară unui algoritm. Vom analiza din acest punct de vedere toți algoritmii prezentați.

v) *Testarea programelor*. Aceasta constă din două faze: depanare (debugging) și trasare (profiling). *Depanarea* este procesul executării unui program pe date de test și corectarea eventualelor erori. După cum afirma însă E. W. Dijkstra, prin depanare putem evidenția prezența erorilor, dar nu și absența lor. O demonstrare a faptului că un program este corect este mai valoroasă decât o mie de teste, deoarece garantează că programul va funcționa corect în *orice* situație. *Trasarea* este procesul executării unui program corect pe diferite date de test, pentru a-i determina timpul de calcul și memoria necesară. Rezultatele obținute pot fi apoi comparate cu analiza anterioară a algoritmului.

Această enumerare servește fixării cadrului general pentru problemele abordate în carte: ne vom concentra pe domeniile *i)*, *ii)* și *iv)*.

Vom începe cu un exemplu de algoritm. Este vorba de o metodă, cam ciudată la prima vedere, de înmulțire a două numere. Se numește "înmulțirea a la russe".

Vom scrie deînmulțitul și înmulțitorul (de exemplu 45 și 19) unul lângă altul, formând sub fiecare câte o coloană, conform următoarei reguli: se împarte numărul de sub deînmulțit la 2, ignorând fracțiile, apoi se înmulțește cu 2 numărul

45	19	19
22	38	—
11	76	76
5	152	152
2	304	—
1	608	608
		855

de sub înmulțitor. Se aplică regula, până când numărul de sub deînmulțit este 1. În final, adunăm toate numerele din coloana înmulțitorului care corespund, pe linie, unor numere impare în coloana deînmulțitului. În cazul nostru, obținem: 19 + 76 + 152 + 608 = 855.

Cu toate că pare ciudată, aceasta este tehnica folosită de hardware-ul multor calculatoare. Ea prezintă avantajul că nu este necesar să se memoreze tabla de înmulțire. Totul se rezumă la adunări și înmulțiri/împărțiri cu 2 (acestea din urmă fiind rezolvate printr-o simplă decalare).

Pentru a reprezenta algoritmul, vom utiliza un limbaj simplificat, numit *pseudo-cod*, care este un compromis între precizia unui limbaj de programare și ușurința în exprimare a unui limbaj natural. Astfel, elementele esențiale ale algoritmului nu vor fi ascunse de detalii de programare neimportante în această fază. Dacă sunteți familiarizat cu un limbaj uzual de programare, nu veți avea nici

o dificultate în a înțelege notațiile folosite și în a scrie programul respectiv. Cunoașteți atunci și diferența dintre o funcție și o procedură. În notația pe care o folosim, o funcție va returna uneori un tablou, o mulțime, sau un mesaj. Veți înțelege că este vorba de o scriere mai compactă și în funcție de context veți putea alege implementarea convenabilă. Vom conveni ca parametrii funcțiilor (procedurilor) să fie transmiși *prin valoare*, exceptând tablourile, care vor fi transmise *prin adresa primului element*. Notația folosită pentru specificarea unui parametru de tip tablou va fi diferită, de la caz la caz. Uneori vom scrie, de exemplu:

procedure *proc*1(*T*)

atunci când tipul și dimensiunile tabloului *T* sunt neimportante, sau când acestea sunt evidente din context. Într-un astfel de caz, vom nota cu #*T* numărul de elemente din tabloului *T*. Dacă limitele sau tipul tabloului sunt importante, vom scrie:

procedure *proc*2(*T*[1 .. *n*])

sau, mai general:

procedure *proc*3(*T*[*a* .. *b*])

În aceste cazuri, *n*, *a* și *b* vor fi considerați parametri formali.

De multe ori, vom atribui unor elemente ale unui tablou *T* valorile $\pm\infty$, înțelegând prin acestea două valori numerice extreme, astfel încât pentru oricare alt element $T[i]$ avem $-\infty < T[i] < +\infty$.

Pentru simplitate, vom considera uneori că anumite variabile sunt globale, astfel încât să le putem folosi în mod direct în proceduri.

Iată acum și primul nostru algoritm, cel al înmulțirii “a la russe”:

```
function russe(A, B)
    arrays X, Y
    {inițializare}
    X[1] ← A; Y[1] ← B
    i ← 1 {se construiesc cele două coloane}
    while X[i] > 1 do
        X[i+1] ← X[i] div 2 {div reprezintă împărțirea întreagă}
        Y[i+1] ← Y[i]+Y[i]
        i ← i+1
    {adună numerele Y[i] corespunzătoare numerelor X[i] impare}
    prod ← 0
    while i > 0 do
        if X[i] este impar then prod ← prod+Y[i]
        i ← i−1
    return prod
```

Un programator cu experiență va observa desigur că tablourile X și Y nu sunt de fapt necesare și că programul poate fi simplificat cu ușurință. Acest algoritm poate fi programat deci în mai multe feluri, chiar folosind același limbaj de programare.

Pe lângă algoritmul de înmulțire învățat în școală, iată că mai avem un algoritm care face același lucru. Există mai mulți algoritmi care rezolvă o problemă, dar și mai multe programe care pot descrie un algoritm.

Acest algoritm poate fi folosit nu doar pentru a înmulți pe 45 cu 19, dar și pentru a înmulți orice numere întregi pozitive. Vom numi (45, 19) un *caz* (*instance*). Pentru fiecare algoritm există un *domeniu de definiție* al cazurilor pentru care algoritmul funcționează corect. Orice calculator limitează mărimea cazurilor cu care poate opera. Această limitare nu poate fi însă atribuită algoritmului respectiv. Încă o dată, observăm că există o diferență esențială între programe și algoritmi.

1.2 Eficiența algoritmilor

Ideal este ca, pentru o problemă dată, să găsim mai mulți algoritmi, iar apoi să-l alegem dintre aceștia pe cel optim. Care este însă criteriul de comparație? Eficiența unui algoritm poate fi exprimată în mai multe moduri. Putem analiza *a posteriori* (empiric) comportarea algoritmului după implementare, prin rularea pe calculator a unor cazuri diferite. Sau, putem analiza *a priori* (teoretic) algoritmul, înaintea programării lui, prin determinarea cantitativă a resurselor (timp, memorie etc) necesare *ca o funcție de mărimea cazului considerat.*

Mărimea unui caz x, notată cu $|x|$, corespunde formal numărului de biți necesari pentru reprezentarea lui x, folosind o codificare precis definită și rezonabil de

compactă. Astfel, când vom vorbi despre sortare, $|x|$ va fi numărul de elemente de sortat. La un algoritm numeric, $|x|$ poate fi chiar valoarea numerică a cazului x.

Avantajul analizei teoretice este faptul că ea nu depinde de calculatorul folosit, de limbajul de programare ales, sau de îndemânarea programatorului. Ea salvează timpul pierdut cu programarea și rularea unui algoritm care se dovedește în final ineficient. Din motive practice, un algoritm nu poate fi testat pe calculator pentru cazuri oricât de mari. Analiza teoretică ne permite însă studiul eficienței algoritmului pentru cazuri de orice mărime.

Este posibil să analizăm un algoritm și printr-o metodă *hibridă*. În acest caz, forma funcției care descrie eficiența algoritmului este determinată teoretic, iar valorile numerice ale parametrilor sunt apoi determinate empiric. Această metodă permite o predicție asupra comportării algoritmului pentru cazuri foarte mari, care nu pot fi testate. O extrapolare doar pe baza testelor empirice este foarte imprecisă.

Este natural să întrebăm ce unitate trebuie folosită pentru a exprima eficiența teoretică a unui algoritm. Un răspuns la această problemă este dat de *principiul invarianței*, potrivit căruia două implementări diferite ale aceluiași algoritm nu diferă în eficiență cu mai mult de o constantă multiplicativă. Adică, presupunând că avem două implementări care necesită $t_1(n)$ și, respectiv, $t_2(n)$ secunde pentru a rezolva un caz de mărime n, atunci există întotdeauna o constantă pozitivă c, astfel încât $t_1(n) \leq ct_2(n)$ pentru orice n suficient de mare. Acest principiu este valabil indiferent de calculatorul (de construcție convențională) folosit, indiferent de limbajul de programare ales și indiferent de îndemânarea programatorului (presupunând că acesta nu modifică algoritmul!). Deci, schimbarea calculatorului ne poate permite să rezolvăm o problemă de 100 de ori mai repede, dar numai modificarea algoritmului ne poate aduce o îmbunătățire care să devină din ce în ce mai marcantă pe măsură ce mărimea cazului soluționat crește.

Revenind la problema unității de măsură a eficienței teoretice a unui algoritm, ajungem la concluzia că nici nu avem nevoie de o astfel de unitate: vom exprima eficiența în limitele unei constante multiplicative. Vom spune că un algoritm necesită timp în *ordinul lui t*, pentru o funcție t dată, dacă există o constantă pozitivă c și o implementare a algoritmului capabilă să rezolve fiecare caz al problemei într-un timp de cel mult $ct(n)$ secunde, unde n este mărimea cazului considerat. Utilizarea secundelor în această definiție este arbitrară, deoarece trebuie să modificăm doar constanta pentru a mărgini timpul la $at(n)$ ore, sau $bt(n)$ microsecunde. Datorită principiului invarianței, orice altă implementare a algoritmului va avea aceeași proprietate, cu toate că de la o implementare la alta se poate modifica constanta multiplicativă. În Capitolul 5 vom reveni mai riguros asupra acestui important concept, numit *notație asimptotică*.

Dacă un algoritm necesită timp în ordinul lui n, vom spune că necesită timp *liniar*, iar algoritmul respectiv putem să-l numim *algoritm liniar*. Similar, un algoritm este *pătratic*, *cubic*, *polinomial*, sau *exponențial* dacă necesită timp în ordinul lui n^2, n^3, n^k, respectiv c^n, unde k și c sunt constante.

Un obiectiv major al acestei cărți este analiza teoretică a eficienței algoritmilor. Ne vom concentra asupra criteriului timpului de execuție. Alte resurse necesare (cum ar fi memoria) pot fi estimate teoretic într-un mod similar. Se pot pune și probleme de compromis memorie - timp de execuție.

1.3 Cazul mediu și cazul cel mai nefavorabil

Timpul de execuție al unui algoritm poate varia considerabil chiar și pentru cazuri de mărime identică. Pentru a ilustra aceasta, vom considera doi algoritmi elementari de sortare a unui tablou T de n elemente:

```
procedure insert(T[1 .. n])
    for i ← 2 to n do
        x ← T[i]; j ← i−1
        while j > 0 and x < T[ j] do
            T[ j+1] ← T[ j]
            j ← j−1
        T[ j+1] ← x

procedure select (T[1 .. n])
    for i ← 1 to n−1 do
        minj ← i; minx ← T[i]
        for j ← i+1 to n do
            if T[ j] < minx then   minj ← j
                                   minx ← T[ j]
        T[minj] ← T[i]
        T[i] ← minx
```

Ideea generală a sortării *prin inserție* este să considerăm pe rând fiecare element al șirului și să îl inserăm în subșirul ordonat creat anterior din elementele precedente. Operația de inserare implică deplasarea spre dreapta a unei secvențe. Sortarea *prin selecție* lucrează altfel, plasând la fiecare pas câte un element direct pe poziția lui finală.

Fie U și V două tablouri de n elemente, unde U este deja sortat crescător, iar V este sortat descrescător. Din punct de vedere al timpului de execuție, V reprezintă cazul cel mai nefavorabil iar U cazul cel mai favorabil.

Vom vedea mai târziu că timpul de execuție pentru sortarea prin selecție este pătratic, independent de ordonarea inițială a elementelor. Testul "**if** $T[j] < minx$" este executat de tot atâtea ori pentru oricare dintre cazuri. Relativ micile variații ale timpului de execuție se datorează doar numărului de executări ale atribuirilor din ramura **then** a testului.

La sortarea prin inserție, situația este diferită. Pe de o parte, *insert*(U) este foarte rapid, deoarece condiția care controlează bucla **while** este mereu falsă. Timpul necesar este liniar. Pe de altă parte, *insert*(V) necesită timp pătratic, deoarece bucla **while** este executată de $i-1$ ori pentru fiecare valoare a lui i. (Vom analiza acest lucru în Capitolul 5).

Dacă apar astfel de variații mari, atunci cum putem vorbi de un timp de execuție care să depindă doar de mărimea cazului considerat? De obicei considerăm analiza pentru cel mai nefavorabil caz. Acest tip de analiză este bun atunci când timpul de execuție al unui algoritm este critic (de exemplu, la controlul unei centrale nucleare). Pe de altă parte însă, este bine uneori să cunoaștem timpul *mediu* de execuție al unui algoritm, atunci când el este folosit foarte des pentru cazuri diferite. Vom vedea că timpul mediu pentru sortarea prin inserție este tot pătratic. În anumite cazuri însă, acest algoritm poate fi mai rapid. Există un algoritm de sortare (*quicksort*) cu timp pătratic pentru cel mai nefavorabil caz, dar cu timpul mediu în ordinul lui $n \log n$. (Prin log notăm logaritmul într-o bază oarecare, lg este logaritmul în baza 2, iar ln este logaritmul natural). Deci, pentru cazul mediu, *quicksort* este foarte rapid.

Analiza comportării în medie a unui algoritm presupune cunoașterea a priori a distribuției probabiliste a cazurilor considerate. Din această cauză, analiza pentru cazul mediu este, în general, mai greu de efecuat decât pentru cazul cel mai nefavorabil.

Atunci când nu vom specifica pentru ce caz analizăm un algoritm, înseamnă că eficiența algoritmului nu depinde de acest aspect (ci doar de mărimea cazului).

1.4 Operație elementară

O *operație elementară* este o operație al cărei timp de execuție poate fi mărginit superior de o constantă depinzând doar de particularitatea implementării (calculator, limbaj de programare etc). Deoarece ne interesează timpul de execuție în limita unei constante multiplicative, vom considera doar numărul operațiilor elementare executate într-un algoritm, nu și timpul exact de execuție al operațiilor respective.

Următorul exemplu este testul lui Wilson de primalitate (teorema care stă la baza acestui test a fost formulată inițial de Leibniz în 1682, reluată de Wilson în 1770 și demonstrată imediat după aceea de Lagrange):

```
function Wilson(n)
    {returnează true dacă și numai dacă n este prim}
    if n divide ((n−1)! + 1)   then   return true
                               else   return false
```

Dacă considerăm calculul factorialului și testul de divizibilitate ca operații elementare, atunci eficiența testului de primalitate este foarte mare. Dacă considerăm că factorialul se calculează în funcție de mărimea lui *n*, atunci eficiența testului este mai slabă. La fel și cu testul de divizibilitate.

Deci, este foarte important ce anume definim ca operație elementară. Este oare adunarea o operație elementară? În teorie, nu, deoarece și ea depinde de lungimea operanzilor. Practic, pentru operanzi de lungime rezonabilă (determinată de modul de reprezentare internă), putem să considerăm că adunarea este o operație elementară. Vom considera în continuare că adunările, scăderile, înmulțirile, împărțirile, operațiile modulo (restul împărțirii întregi), operațiile booleene, comparațiile și atribuirile sunt operații elementare.

1.5 De ce avem nevoie de algoritmi eficienți?

Performanțele hardware-ului se dublează la aproximativ doi ani. Mai are sens atunci să investim în obținerea unor algoritmi eficienți? Nu este oare mai simplu să așteptăm următoarea generație de calculatoare?

Să presupunem că pentru rezolvarea unei anumite probleme avem un algoritm exponențial și un calculator pe care, pentru cazuri de mărime n, timpul de rulare este de $10^{-4} \times 2^n$ secunde. Pentru $n = 10$, este nevoie de 1/10 secunde. Pentru $n = 20$, sunt necesare aproape 2 minute. Pentru $n = 30$, o zi nu este de ajuns, iar pentru $n = 38$, chiar și un an ar fi insuficient. Cumpărăm un calculator de 100 de ori mai rapid, cu timpul de rulare de $10^{-6} \times 2^n$ secunde. Dar și acum, pentru $n = 45$, este nevoie de mai mult de un an! În general, dacă în cazul mașinii vechi într-un timp anumit se putea rezolva problema pentru cazul n, pe noul calculator, în acest timp, se poate rezolva cazul n+7.

Să presupunem acum că am găsit un algoritm cubic care rezolvă, pe calculatorul vechi, cazul de mărime n în $10^{-2} \times n^3$ secunde. În Figura 1.1, putem urmări cum evoluează timpul de rulare în funcție de mărimea cazului. Pe durata unei zile, rezolvăm acum cazuri mai mari decât 200, iar în aproximativ un an am putea rezolva chiar cazul $n = 1500$. Este mai profitabil să investim în noul algoritm decât într-un nou hardware. Desigur, dacă ne permitem să investim atât în software cât și în hardware, noul algoritm poate fi rulat și pe noua mașină. Curba $10^{-4} \times n^3$ reprezintă această din urmă situație.

Pentru cazuri de mărime mică, uneori este totuși mai rentabil să investim într-o

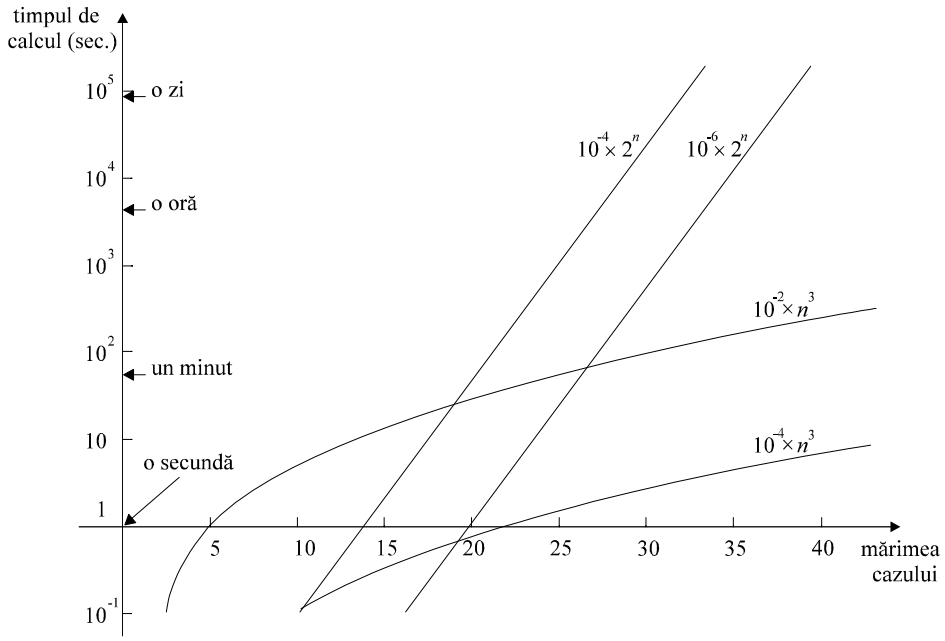


Figura 1.1 Algoritmi sau hardware?

nouă mașină, nu și într-un nou algoritm. Astfel, pentru $n = 10$, pe mașina veche, algoritmul nou necesită 10 secunde, adică de o sută de ori mai mult decât algoritmul vechi. Pe vechiul calculator, algoritmul nou devine mai performant doar pentru cazuri mai mari sau egale cu 20.

1.6 Exemple

Poate că vă întrebați dacă este într-adevăr posibil să accelerăm atât de spectaculos un algoritm. Răspunsul este afirmativ și vom da câteva exemple.

1.6.1 Sortare

Algoritmii de sortare prin inserție și prin selecție necesită timp pătratic, atât în cazul mediu, cât și în cazul cel mai nefavorabil. Cu toate că acești algoritmi sunt excelenți pentru cazuri mici, pentru cazuri mari avem algoritmi mai eficienți. În capitolele următoare vom analiza și alți algoritmi de sortare: *heapsort*, *mergesort*, *quicksort*. Toți aceștia necesită un timp mediu în ordinul lui $n \log n$, iar *heapsort* și *mergesort* necesită timp în ordinul lui $n \log n$ și în cazul cel mai nefavorabil.

Pentru a ne face o idee asupra diferenței dintre un timp pătratic și un timp în ordinul lui $n \log n$, vom menționa că, pe un anumit calculator, *quicksort* a reușit să sorteze în 30 de secunde 100.000 de elemente, în timp ce sortarea prin inserție ar fi durat, pentru același caz, peste nouă ore. Pentru un număr mic de elemente însă, eficiența celor două sortări este asemănătoare.

1.6.2 Calculul determinanților

Fie $\det(M)$ determinantul matricii

$$M = (a_{ij})_{i,\, j\, =\, 1,\, \ldots,\, n}$$

și fie M_{ij} submatricea de $(n-1) \times (n-1)$ elemente, obținută din M prin ștergerea celei de-a i-a linii și celei de-a j-a coloane. Avem binecunoscuta definiție recursivă

$$\det(M) = \sum_{j=1}^{n} (-1)^{j+1} a_{1j} \det(M_{1j})$$

Dacă folosim această relație pentru a evalua determinantul, obținem un algoritm cu timp în ordinul lui $n!$, ceea ce este mai rău decât exponențial. O altă metodă clasică, eliminarea Gauss-Jordan, necesită timp cubic. Pentru o anumită

implementare s-a estimat că, în cazul unei matrici de 20 × 20 elemente, în timp ce algoritmul Gauss-Jordan durează 1/20 secunde, algoritmul recursiv ar dura mai mult de 10 milioane de ani!

Nu trebuie trasă de aici concluzia că algoritmii recursivi sunt în mod necesar neperformanți. Cu ajutorul algoritmului recursiv al lui Strassen, pe care îl vom studia și noi în Secțiunea 7.8, se poate calcula det(M) într-un timp în ordinul lui $n^{\lg 7}$, unde $\lg 7 \cong 2{,}81$, deci mai eficient decât prin eliminarea Gauss-Jordan.

1.6.3 Cel mai mare divizor comun

Un prim algoritm pentru aflarea celui mai mare divizor comun al întregilor pozitivi m și n, notat cu cmmdc(m, n), se bazează pe definiție:

```
function cmmdc-def (m, n)
    i ← min(m, n) + 1
    repeat i ← i–1 until i divide pe m și n
    return i
```

Timpul este în ordinul diferenței dintre min(m, n) și cmmdc(m, n).

Există, din fericire, un algoritm mult mai eficient, care nu este altul decât celebrul algoritm al lui Euclid.

```
function Euclid(m, n)
    if n = 0  then  return m
              else  return Euclid(n, m mod n)
```

Prin m **mod** n notăm restul împărțirii întregi a lui m la n. Algoritmul funcționează pentru orice întregi nenuli m și n, având la bază cunoscuta proprietate

$$\text{cmmdc}(m, n) = \text{cmmdc}(n, m \bmod n)$$

Timpul este în ordinul logaritmului lui min(m, n), chiar și în cazul cel mai nefavorabil, ceea ce reprezintă o îmbunătățire substanțială față de algoritmul precedent. Pentru a fi exacți, trebuie să menționăm că algoritmul originar al lui Euclid (descris în "*Elemente*", aprox. 300 a.Ch.) operează prin scăderi succesive, și nu prin împărțire. Interesant este faptul că acest algoritm se pare că provine dintr-un algoritm și mai vechi, datorat lui Eudoxus (aprox. 375 a.Ch.).

1.6.4 Numerele lui Fibonacci

Șirul lui Fibonacci este definit prin următoarea recurență:

$$\begin{cases} f_0 = 0;\ f_1 = 1 \\ f_n = f_{n-1} + f_{n-2} \quad \text{pentru} \quad n \geq 2 \end{cases}$$

Acest celebru șir a fost descoperit în 1202 de către Leonardo Pisano (Leonardo din Pisa), cunoscut sub numele de Leonardo Fibonacci. Cel de-al n-lea termen al șirului se poate obține direct din definiție:

```
function fib1(n)
    if n < 2   then   return n
               else   return fib1(n−1) + fib1(n−2)
```

Această metodă este foarte ineficientă, deoarece recalculează de mai multe ori aceleași valori. Vom arăta în Secțiunea 5.3.1 că timpul este în ordinul lui ϕ^n, unde $\phi = (1+\sqrt{5})/2$ este *secțiunea de aur*, deci este un timp exponențial.

Iată acum o altă metodă, mai performantă, care rezolvă aceeași problemă într-un timp liniar.

```
function fib2(n)
    i ← 1; j ← 0
    for k ← 1 to n do  j ← i + j
                       i ← j − i
    return j
```

Mai mult, există și un algoritm cu timp în ordinul lui $\log n$, algoritm pe care îl vom argumenta însă abia în Capitolul 7:

```
function fib3(n)
    i ← 1; j ← 0; k ← 0; h ← 1
    while n > 0 do
        if n este impar then   t ← jh
                               j ← ih+jk+t
                               i ← ik+t
        t ← h^2
        h ← 2kh+t
        k ← k^2+t
        n ← n div 2
    return j
```

Vă recomandăm să comparați acești trei algoritmi, pe calculator, pentru diferite valori ale lui n.

1.7 Exerciții

1.1 Aplicați algoritmii *insert* și *select* pentru cazurile $T = [1, 2, 3, 4, 5, 6]$ și $U = [6, 5, 4, 3, 2, 1]$. Asigurați-vă că ați înțeles cum funcționează.

1.2 Înmulțirea "a la russe" este cunoscută încă din timpul Egiptului antic, fiind probabil un algoritm mai vechi decât cel al lui Euclid. Încercați să înțelegeți raționamentul care stă la baza acestui algoritm de înmulțire.

Indicație: Faceți legătura cu reprezentarea binară.

1.3 În algoritmul *Euclid*, este important ca $n \geq m$?

1.4 Elaborați un algoritm care să returneze cel mai mare divizor comun a trei întregi nenuli.

Soluție:

```
function Euclid-trei(m, n, p)
    return Euclid(m, Euclid(n, p))
```

1.5 Programați algoritmul *fib*1 în două limbaje diferite și rulați comparativ cele două programe, pe mai multe cazuri. Verificați dacă este valabil principiul invarianței.

1.6 Elaborați un algoritm care returnează cel mai mare divizor comun a doi termeni de rang oarecare din șirul lui Fibonacci.

Indicație: Un algoritm eficient se obține folosind următoarea proprietate[*], valabilă pentru oricare doi termeni ai șirului lui Fibonacci:

$$\text{cmmdc}(f_m, f_n) = f_{\text{cmmdc}(m, n)}$$

1.7 Fie matricea $M = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$. Calculați produsul vectorului (f_{n-1}, f_n) cu matricea M^m, unde f_{n-1} și f_n sunt doi termeni consecutivi oarecare ai șirului lui Fibonacci.

---

[*] Această surprinzătoare proprietate a fost descoperită în 1876 de Lucas.

2. Programare orientată pe obiect

Deşi această carte este dedicată în primul rând analizei şi elaborării algoritmilor, am considerat util să folosim numeroşii algoritmi care sunt studiaţi ca un pretext pentru introducerea elementelor de bază ale programării orientate pe obiect în limbajul C++. Vom prezenta în capitolul de faţă noţiuni fundamentale legate de obiecte, limbajul C++ şi de abstractizarea datelor în C++, urmând ca, pe baza unor exemple detaliate, să conturăm în capitolele următoare din ce în ce mai clar tehnica programării orientate pe obiect. Scopul urmărit este de a surprinde acele aspecte strict necesare formării unei impresii juste asupra programării orientate pe obiect în limbajul C++, şi nu de a substitui cartea de faţă unui curs complet de C++.

2.1 Conceptul de obiect

Activitatea de programare a calculatoarelor a apărut la sfârşitul anilor ‘40. Primele programe au fost scrise în limbaj maşină şi de aceea depindeau în întregime de arhitectura calculatorului pentru care erau concepute. Tehnicile de programare au evoluat apoi în mod natural spre o tot mai netă separare între conceptele manipulate de programe şi reprezentările acestor concepte în calculator.

În faţa complexităţii crescânde a problemelor care se cereau soluţionate, structurarea programelor a devenit indispensabilă. Şcoala de programare Algol a propus la începutul anilor ‘60 o abordare devenită între timp clasică. Conform celebrei ecuaţii a lui Niklaus Wirth:

algoritmi + structuri de date = programe

un program este format din două părţi total separate: un ansamblu de proceduri şi un ansamblu de date asupra cărora acţionează procedurile. Procedurile sunt privite ca şi cutii negre, fiecare având de rezolvat o anumită sarcină (de făcut anumite prelucrări). Această modalitate de programare se numeşte *programare dirijată de prelucrări*. Evoluţia calculatoarelor şi a problemelor de programare a făcut ca în aproximativ zece ani programarea dirijată de prelucrări să devină ineficientă. Astfel, chiar dacă un limbaj ca Pascal-ul permite o bună structurare a programului în proceduri, este posibil ca o schimbare relativ minoră în structura datelor să provoace o dezorganizare majoră a procedurilor.

Inconvenientele programării dirijate de prelucrări sunt eliminate prin *încapsularea* datelor și a procedurilor care le manipulează într-o singură entitate numită *obiect*. Lumea exterioară obiectului are acces la datele sau procedurile lui doar prin intermediul unor operații care constituie *interfața* obiectului. Programatorul nu este obligat să cunoască reprezentarea fizică a datelor și procedurilor utilizate, motiv pentru care poate trata obiectul ca pe o cutie neagră cu un comportament bine precizat. Această caracteristică permite realizarea unor *tipuri abstracte de date*. Este vorba de obiecte înzestrate cu o interfață prin care se specifică interacțiunile cu exteriorul, singura modalitate de a comunica cu un astfel de obiect fiind invocarea interfeței sale. În terminologia specifică programării orientate pe obiect, procedurile care formează interfața unui obiect se numesc *metode*. Obiectul este singurul responsabil de maniera în care se efectuează operațiile asupra lui. Apelul unei metode este doar o cerere, un *mesaj* al apelantului care solicită executarea unei anumite acțiuni. Obiectul poate refuza să o execute, sau, la fel de bine, o poate transmite unui alt obiect. În acest context, programarea devine *dirijată de date*, și nu de prelucrările care trebuie realizate.

Utilizarea consecventă a obiectelor conferă programării următoarele calități:

- *Abstractizarea datelor*. Nu este nevoie de a cunoaște implementarea și reprezentarea internă a unui obiect pentru a-i adresa mesaje. Obiectul decide singur maniera de execuție a operației cerute în functie de implementarea fizică. Este posibilă supraîncărcarea metodelor, în sensul că la aceleași mesaje, obiecte diferite răspund în mod diferit. De exemplu, este foarte comod de a desemna printr-un simbol unic, +, adunarea întregilor, concatenarea șirurilor de caractere, reuniunea mulțimilor etc.
- *Modularitate*. Structura programului este determinată în mare măsură de obiectele utilizate. Schimbarea definițiilor unor obiecte se poate face cu un minim de implicații asupra celorlalte obiecte utilizate în program.
- *Flexibilitate*. Un obiect este definit prin comportamentul său grație existenței unei interfețe explicite. El poate fi foarte ușor introdus într-o bibliotecă pentru a fi utilizat ca atare, sau pentru a construi noi tipuri prin *moștenire*, adică prin specializare și compunere cu obiecte existente.
- *Claritate*. Încapsularea, posibilitatea de supraîncărcare și modularitatea întăresc claritatea programelor. Detaliile de implementare sunt izolate de lumea exterioară, numele metodelor pot fi alese cât mai natural posibil, iar interfețele specifică precis și detaliat modul de utilizare al obiectului.

2.2 Limbajul C++

Toate limbajele de nivel înalt, de la FORTRAN la LISP, permit adaptarea unui stil de programare orientat pe obiect, dar numai câteva oferă mecanismele pentru

utilizarea directă a obiectelor. Din acest punct de vedere, menționăm două mari categorii de limbaje:

- Limbaje care oferă doar facilități de abstractizarea datelor și încapsulare, cum sunt Ada și Modula-2. De exemplu, în Ada, datele și procedurile care le manipulează pot fi grupate într-un pachet (package).
- Limbaje orientate pe obiect, care adaugă abstractizării datelor noțiunea de moștenire.

Deși definițiile de mai sus restrâng mult mulțimea limbajelor calificabile ca "orientate pe obiect", aceste limbaje rămân totuși foarte diverse, atât din punct de vedere al conceptelor folosite, cât și datorită modului de implementare. S-au conturat trei mari familii, fiecare accentuând un anumit aspect al noțiunii de obiect: limbaje de clase, limbaje de cadre (frames) și limbaje de tip actor.

Limbajul C++* aparține familiei limbajelor de clase. O *clasă* este un tip de date care descrie un ansamblu de obiecte cu aceeași structură și același comportament. Clasele pot fi îmbogățite și completate pentru a defini alte familii de obiecte. În acest mod se obțin ierarhii de clase din ce în ce mai specializate, care *moștenesc* datele și metodele claselor din care au fost create. Din punct de vedere istoric primele limbaje de clase au fost Simula (1973) și Smalltalk-80 (1983). Limbajul Simula a servit ca model pentru o întregă linie de limbaje caracterizate printr-o organizare statică a tipurilor de date.

Să vedem acum care sunt principalele deosebiri dintre limbajele C și C++, precum și modul în care s-au implementat intrările/ieșirile în limbajul C++.

2.2.1 Diferențele dintre limbajele C și C++

Limbajul C, foarte lejer în privința verificării tipurilor de date, lasă programatorului o libertate deplină. Această libertate este o sursă permanentă de erori și de efecte colaterale foarte dificil de depanat. Limbajul C++ a introdus o verificare foarte strictă a tipurilor de date. În particular, apelul oricarei funcții trebuie precedat de *declararea* funcției respective. Pe baza declarațiilor, prin care se specifică numărul și tipul parametrilor formali, parametrii efectivi poat fi verificați în momentul compilării apelului. În cazul unor nepotriviri de tipuri, compilatorul încearcă realizarea corespondenței (*matching*) prin invocarea unor conversii, semnalând eroare doar dacă nu găsește nici o posibilitate.

```
float maxim( float, float );
float x = maxim( 3, 2.5 );
```

* Limbaj dezvoltat de Bjarne Stroustrup la începutul anilor '80, în cadrul laboratoarelor Bell de la AT&T, ca o extindere orientată pe obiect a limbajului C.

În acest exemplu, funcția `maxim()` este declarată ca o funcție de tip `float` cu doi parametri tot de tip `float`, motiv pentru care constanta întreagă `3` este convertită în momentul apelului la tipul `float`. Declarația unei funcții constă în *prototipul funcției*, care conține tipul valorii returnate, numele funcției, numărul și tipul parametrilor. Diferența dintre *definiție* și declarație – noțiuni valabile și pentru variabile – constă în faptul că definiția este o declarație care provoacă și rezervare de spațiu sau generare de cod. Declararea unei variabile se face prin precedarea obligatorie a definiției de cuvântul cheie `extern`. Și o declarație de funcție poate fi precedată de cuvântul cheie `extern`, accentuând astfel că funcția este definită altundeva.

Definirea unor funcții foarte mici, pentru care procedura de apel tinde să dureze mai mult decât executarea propriu-zisă, se realizează în limbajul C++ prin funcțiile `inline`.

```
inline float maxim( float x, float y ) {
  putchar( 'r' ); return x > y? x: y;
}
```

Specificarea `inline` este doar orientativă și indică compilatorului că este preferabil de a înlocui fiecare apel cu corpul funcției apelate. Expandarea unei funcții `inline` nu este o simplă substituție de text în progamul sursă, deoarece se realizează prin păstrarea semanticii apelului, deci inclusiv a verificării corespondenței tipurilor parametrilor efectivi.

Mecanismul de verificare a tipului lucrează într-un mod foarte flexibil, permițând atât existența funcțiilor cu un număr variabil de argumente, cât și a celor *supraîncărcate*. Supraîncărcarea permite existența mai multor funcții cu același nume, dar cu paremetri diferiți. Eliminarea ambiguității care apare în momentul apelului se rezolvă pe baza numărului și tipului parametrilor efectivi. Iată, de exemplu, o altă funcție `maxim()`:

```
inline int maxim( int x, int y ) {
  putchar( 'i' ); return x > y? x: y;
}
```

(Prin apelarea funcției `putchar()`, putem afla care din cele două funcții `maxim()` este efectiv invocată).

În limbajul C++ nu este obligatorie definirea variabilelor locale strict la începutul blocului de instrucțiuni. În exemplul de mai jos, tabloul `buf` și întregul `i` pot fi utilizate din momentul definirii și până la sfârșitul blocului în care au fost definite.

```
#define DIM 5

void f( ) {
  int buf[ DIM ];

  for ( int i = 0; i < DIM; )
    buf[ i++ ] = maxim( i, DIM - i );
  while ( --i )
    printf( "%3d ", buf[ i ] );
}
```

În legătură cu acest exemplu, să mai notăm şi faptul că instrucţiunea `for` permite chiar definirea unor variabile (variabila `i` în cazul nostru). Variabilele definite în instrucţiunea `for` pot fi utilizate la nivelul blocului acestei instrucţiuni şi după terminarea executării ei.

Deşi transmiterea parametrilor în limbajul C se face numai prin valoare, limbajul C++ autorizează în egală măsură şi transmiterea prin *referinţă*. Referinţele, indicate prin caracterul `&`, permit accesarea în scriere a parametrilor efectivi, fără transmiterea lor prin adrese. Iată un exemplu în care o procedură interschimbă (swap) valorile argumentelor.

```
void swap( float& a, float& b ) {
  float tmp = a; a = b; b = tmp;
}
```

Referinţele evită duplicarea provocată de transmiterea parametrilor prin valoare şi sunt utile mai ales în cazul transmiterii unor structuri. De exemplu, presupunând existenţa unei structuri de tip `struct punct`,

```
struct punct {
  float x;     /* coordonatele unui */
  float y;     /* punct din plan    */
};
```

următoarea funcţie transformă un punct în simetricul lui faţă de cea de a doua bisectoare.

```
void sim2( struct punct& p ) {
  swap( p.x, p.y ); // p.x si p.y se transmit prin
                    // referinta si nu prin valoare
  p.x = -p.x; p.y = -p.y;
}
```

Parametrii de tip referinţă pot fi protejaţi de modificări accidentale prin declararea lor `const`.

```
void print( const struct punct& p ) {
  // compilatorul interzice orice tentativa
  // de a modifica variabila p
  printf( "(%4.1f, %4.1f) ", p.x, p.y );
}
```

Caracterele `//` indică faptul că restul liniei curente este un comentariu. Pe lângă această modalitate nouă de a introduce comentarii, limbajul C++ a preluat din limbajul C și posibiliatea încadrării lor între `/*` și `*/`.

Atributul `const` poate fi asociat nu numai parametrilor formali, ci și unor definiții de variabile, a căror valoare este specificată în momentul compilării. Aceste variabile sunt variabile read-only (constante), deoarece nu mai pot fi modificate ulterior. În limbajul C, constantele pot fi definite doar prin intermediul directivei `#define`, care este o sursă foarte puternică de erori. Astfel, în exemplul de mai jos, constanta întreagă `dim` este o variabilă propriu-zisă accesibilă doar în funcția `g()`. Dacă ar fi fost definită prin `#define` (vezi simbolul `DIM` utilizat în funcția `f()` de mai sus) atunci orice identificator `dim`, care apare după directiva de definire și până la sfârșitul fișierului sursă, este înlocuit cu valoarea respectivă, fără nici un fel de verificări sintactice.

```
void g( ) {
  const int dim = 5;
  struct punct buf[ dim ];

  for ( int i = 0; i < dim; i++ ) {
    buf[ i ].x = i;
    buf[ i ].y = dim / 2. - i;

    sim2(  buf[ i ] );
    print( buf[ i ] );
  }
}
```

Pentru a obține un prim program în C++, nu avem decât să adăugăm obișnuitul

```
#include <stdio.h>
```

precum și funcția `main()`

```
int main( ) {
  puts( "\n main." );

  puts( "\n  f( )" ); f( );
  puts( "\n  g( )" ); g( );
```

```
  puts( "\n ---\n" );
  return 0;
}
```

Rezultatele obținute în urma rulării acestui program:

```
r
 main.

  f( )
iiiii  4   3   3   4
  g( )
(-2.5, -0.0) (-1.5, -1.0) (-0.5, -2.0)
( 0.5, -3.0) ( 1.5, -4.0)
 ---
```

suprind prin faptul că funcția `float maxim( float, float )` este invocată înaintea funcției `main()`. Acest lucru este normal, deoarece variabila `x` trebuie inițializată înaintea lansării în execuție a funcției `main()`.

2.2.2 Intrări/ieșiri în limbajul C++

Limbajul C++ permite definirea tipurilor abstracte de date prin intermediul claselor. Clasele nu sunt altceva decât generalizări ale structurilor din limbajul C. Ele conțin date membre, adică variabile de tipuri predefinite sau definite de utilizator prin intermediul altor clase, precum și funcții membre, reprezentând metodele clasei.

Cele mai utilizate clase C++ sunt cele prin care se realizează intrările și ieșirile. Reamintim că în limbajul C, intrările și ieșirile se fac prin intermediul unor funcții de bibliotecă cum sunt `scanf()` și `printf()`, funcții care permit citirea sau scrierea numai a datelor (variabilelor) de tipuri predefinite (`char`, `int`, `float` etc.). Biblioteca standard asociată oricărui compilator C++, conține ca suport pentru operațiile de intrare și ieșire nu simple funcții, ci un set de clase adaptabile chiar și unor tipuri noi, definite de utilizator. Această bibliotecă este un exemplu tipic pentru avantajele oferite de programarea orientată pe obiect. Pentru fixarea ideilor, vom folosi un program care determină termenul de rang *n* al șirului lui Fibonacci prin algoritmul *fib*2 din Secțiunea 1.6.4.

```
#include <iostream.h>

long fib2( int n ) {
  long i = 1, j = 0;

  for ( int k = 0; k++ < n; j = i + j, i = j - i );
  return j;
}

int main( ) {
  cout << "\nTermenul sirului lui Fibonacci de rang ... ";

  int n;
  cin >> n;

  cout << " este " << fib2( n );
  cout << '\n';

  return 0;
}
```

Biblioteca standard C++ conține definițiile unor clase care reprezintă diferite tipuri de *fluxuri de comunicație* (`stream`-uri). Fiecare flux poate fi *de intrare*, *de ieșire*, sau de *intrare/ieșire*. Operația primară pentru fluxul de ieșire este *inserarea* de date, iar pentru cel de ieșire este *extragerea* de date. Fișierul prefix (header) `iostream.h` conține declarațiile fluxului de intrare (clasa `istream`), ale fluxului de ieșire (clasa `ostream`), precum și declarațiile obiectelor `cin` și `cout`:

```
extern istream cin;
extern ostream cout;
```

Operațiile de inserare și extragere sunt realizate prin funcțiile membre ale claselor `ostream` și `istream`. Deoarece limbajul C++ permite existența unor funcții care supraîncarcă o parte din operatorii predefiniți, s-a convenit ca inserarea să se facă prin supraîncarcarea operatorului de decalare la stânga `<<`, iar extragerea prin supraîncărcarea celui de decalare la dreapta `>>`. Semnificația secvenței de instrucțiuni

```
cin >> n;
cout << " este " << fib2( n );
```

este deci următoarea: se citește valoarea lui `n`, apoi se afișează șirul `" este "` urmat de valoarea returnată de funcția `fib2()`.

Fluxurile de comunicație `cin` și `cout` lucrează în mod implicit cu terminalul utilizatorului. Ca și pentru programele scrise în C, este posibilă redirectarea lor spre alte dispozitive sau în diferite fișiere, în funcție de dorința utilizatorului. Pentru sistemele de operare UNIX și DOS, redirectările se indică adăugând

comenzii de lansare în execuție a programului, argumente de forma `>nume-fisier-iesire`, sau `<nume-fisier-intrare`. În `iostream.h` mai este definit încă un flux de ieșire numit `cerr`, utilizabil pentru semnalarea unor condiții de excepție. Fluxul `cerr` este legat de terminalul utilizatorului și nu poate fi redirectat.

Operatorii de inserare (`<<`) și extragere (`>>`) sunt, la rândul lor, supraîncărcați astfel încât operandul drept să poată fi de orice tip predefinit. De exemplu, în instrucțiunea

```
cout << " este " << fib2( n );
```

se va apela operatorul de inserare cu argumentul drept de tip `char*`. Acest operator, ca și toți operatorii de inserare și extragere, returnează operandul stâng, adică `stream`-ul. Astfel, invocarea a doua oară a operatorului de inserare are sens, de acesată dată alegându-se cel cu argumentul drept de tip `long`. În prezent, biblioteca standard de intrare/ieșire are în jur de 4000 de linii de cod, și conține 15 alternative pentru fiecare din operatorii `<<` și `>>`. Programatorul poate supraîncărca în continuare acești operatori pentru propriile tipuri.

2.3 Clase în limbajul C++

Rulând programul pentru determinarea termenilor din șirul lui Fibonacci cu valori din ce în ce mai mari ale lui `n`, se observă că rezultatele nu mai pot fi reprezentate într-un `int`, `long` sau `unsigned long`. Soluția care se impune este de a limita rangul `n` la valori rezonabile reprezentării alese. Cu alte cuvinte, `n` nu mai este de tip `int`, ci de un tip care limitează valorile întregi la un anumit interval. Vom elabora o clasă corespunzătoare acestui tip de întregi, clasă utilă multor programe în care se cere menținerea unei valori între anumite limite.

Clasa se numește `intErval`, și va fi implementată în două variante. Prima variantă este realizată în limbajul C. Nu este o clasă propriu-zisă, ci o structură care confirmă faptul că orice limbaj permite adaptarea unui stil de programare orientat pe obiect și scoate în evidență inconvenientele generate de lipsa mecanismelor de manipulare a obiectelor. A doua variantă este scrisă în limbajul C++. Este un adevărat tip abstract ale cărui calități sunt și mai bine conturate prin comparația cu (pseudo) tipul elaborat în C.

2.3.1 Tipul `intErval` în limbajul C

Reprezentarea internă a tipului conține trei membri de tip întreg: marginile intervalului și valoarea propriu-zisă. Le vom grupa într-o structură care, prin intermediul instrucțiunii `typedef`, devine sinonimă cu `intErval`.

```
typedef struct {
  int min; /* marginea inferioara a intervalului */
  int max; /* marginea superioara a intervalului */
  int v;   /* valoarea, min <= v, v < max        */
} intErval;
```

Variabilele (obiectele) de tip `intErval` se definesc folosind sintaxa uzuală din limbajul C.

```
intErval numar = { 80, 32, 64 };
intErval indice, limita;
```

Efectul acestor definiții constă în rezervarea de spațiu pentru fiecare din datele membre ale obiectelor `numar`, `indice` și `limita`. În plus, datele membre din `numar` sunt inițializate cu valorile 80 (`min`), 32 (`max`) și 64 (`v`). Inițializarea, deși corectă din punct de vedere sintactic, face imposibilă funcționarea tipului `intErval`, deoarece marginea inferioară nu este mai mică decât cea superioară. Deocamdată nu avem nici un mecanism pentru a evita astfel de situații.

Pentru manipularea obiectelor de tip `intErval`, putem folosi atribuiri la nivel de structură:

```
limita = numar;
```

Astfel de atribuiri se numesc *atribuiri membru cu membru*, deoarece sunt realizate între datele membre corespunzătoare celor două obiecte implicate în atribuire.

O altă posibilitate este accesul direct la membri:

```
indice.min = 32; indice.max = 64;
indice.v = numar.v + 1;
```

Selectarea directă a membrilor încalcă proprietățile fundamentale ale obiectelor. Reamintim că un obiect este manipulat exclusiv prin interfața sa, structura lui internă fiind în general inaccesibilă.

Comportamentul obiectelor este realizat printr-un set de metode implementate în limbajul C ca funcții. Pentru `intErval`, acestea trebuie să permită în primul rând selectarea, atât în scriere cât și în citire, a valorii propriu-zise date de membrul `v`. Funcția de scriere `atr()` verifică încadrarea noii valori în domeniul admisibil, iar

funcția de citire `val()` pur și simplu returnează valoarea `v`. Practic, aceste două funcții implementează o formă de încapsulare, izolând reprezentarea internă a obiectului de restul programului.

```
int atr( intErval *pn, int i ) {
  return pn->v = verDom( *pn, i );
}

int val( intErval n ) {
  return n.v;
}
```

Funcția `verDom()` verifică încadrarea în domeniul admisibil:

```
int verDom( intErval n, int i ) {
  if ( i < n.min || i >= n.max ) {
    fputs( "\n\nintErval -- valoare exterioara.\n\n", stderr);
    exit( 1 );
  }
  return i;
}
```

Utilizând consecvent cele două metode ale tipului `intErval`, obținem obiecte ale căror valori sunt cu certitudine între limitele admisibile. De exemplu, utilizând metodele `atr()` și `val()`, instrucțiunea

```
indice.v = numar.v + 1;
```

devine

```
atr( &indice, val( numar ) + 1 );
```

Deoarece `numar` are valoarea 64, iar domeniul `indice`-lui este 32, ..., 64, instrucțiunea de mai sus semnalează depășirea domeniului variabilei `indice` și provoacă terminarea executării programului.

Această implementare este departe de a fi completă și comod de utilizat. Nu ne referim acum la aspecte cum ar fi citirea (sau scrierea) obiectelor de tip `intErval`, operație rezolvabilă printr-o funcție de genul

```
void cit( intErval *pn ) {
  int i;
  scanf( "%d", &i );
  atr( pn, i );
}
```

ci la altele, mult mai delicate, cum ar fi:

I_1 Evitarea unor inițializări eronate din punct de vedere semantic și interzicerea utilizării obiectelor neinițializate:

```
intErval numar = {80,32,64}; // obiect incorect initializat
intErval indice, limita;     // obiecte neinitializate
```

I_2 Interzicerea modificării necontrolate a datelor membre:

```
indice.v = numar.v + 1;
```

I_3 Sintaxa foarte încărcată, diferită de sintaxa obișnuită în manipularea tipurilor întregi predefinite.

În concluzie, această implementare, în loc să ne simplifice activitatea de programare, mai mult a complicat-o. Cauza nu este însă conceperea greșită a tipului `intErval`, ci lipsa facilităților de manipulare a obiectelor din limbajul C.

2.3.2 Tipul `intErval` în limbajul C++

Clasele se obțin prin completarea structurilor uzuale din limbajul C cu setul de funcții necesar implementării interfeței obiectului. În plus, pentru realizarea izolării reprezentării interne de restul programului, fiecărui membru i se asociază nivelul de încapsulare `public` sau `private`. Un membru `public` corespunde, din punct de vedere al nivelului de accesibilitate, membrilor structurilor din limbajul C. Membrii `private` sunt accesibili doar în *domeniul clasei*, adică în clasa propriu-zisă și în toate funcțiile membre. În clasa `intErval`, membrii publici sunt doar funcțiile `atr()` și `val()`, iar membrii `verDom()`, `min`, `max` și `v` sunt privați.

```
class intErval {
public:
  int atr( int );
  int val( ) { return v; }

private:
  int verDom( int );

  int min, max;
  int v;
};
```

Obiectele de tip `intErval` se definesc ca și în limbajul C.

```
intErval numar;
intErval indice, limita;
```

Aceste obiecte pot fi atribuite între ele (fiind structuri atribuirea se va face membru cu membru):

```
limita = numar;
```

și pot fi inițializate (tot membru cu membru) cu un obiect de același tip:

```
intErval cod = numar;
```

Selectarea membrilor se face prin notațiile utilizate pentru structuri. De exemplu, după executarea instrucțiunii

```
indice.atr( numar.val( ) + 1 );
```

valoarea obiectului `indice` va fi valoarea obiectului `numar`, incrementată cu 1. Această operație poate fi descrisă și prin intrucțiunea

```
indice.v = numar.v + 1;
```

care, deși corectă din punct de vedere sintactic, este incorectă semantic, deoarece `v` este un membru `private`, deci inaccesibil prin intermediul obiectelor `indice` și `numar`.

După cum se observă, au dispărut argumentele de tip `intErval*` și `intErval` ale funcțiilor `atr()`, respectiv `val()`. Cauza este faptul că funcțiile membre au un argument implicit, concretizat în *obiectul invocator*, adică obiectul care selectează funcția. Este o convenție care întărește și mai mult atributul de funcție membră (metodă) deoarece permite invocarea unei astfel de funcții numai prin obiectul respectiv.

Definirea funcțiilor membre se poate face fie în corpul clasei, fie în exteriorul acestuia. Funcțiile definite în corpul clasei sunt considerate implicit `inline`, iar pentru cele definite în exteriorul corpului se impune precizarea statutului de funcție membră. Înainte de a defini funcțiile `atr()` și `verDom()`, să observăm că funcția `val()`, definită în corpul clasei `intErval`, încalcă de două ori cele precizate până aici. În primul rând, nu selectează membrul `v` prin intermediul unui obiect, iar în al doilea rând, `v` este privat! Dacă funcția `val()` ar fi fost o funcție obișnuită, atunci observația ar fi fost cât se poate de corectă. Dar `val()` este funcție membră și atunci:

- Nu poate fi apelată decât prin intermediul unui obiect invocator și toți membrii utilizați sunt membrii obiectului invocator.

- Încapsularea unui membru funcționează doar în exteriorul domeniului clasei. Funcțiile membre fac parte din acest domeniu și au acces la toți membrii, indiferent de nivelul lor de încapsulare.

Specificarea atributului de funcție membră se face precedând numele funcției de operatorul domeniu `::` și de numele domeniului, care este chiar numele clasei. Pentru asigurarea consistenței clasei, funcțiile membre definite în exterior trebuie obligatoriu declarate în corpul clasei.

```
int intErval::verDom( int i ) {
  if ( i < min || i >= max ) {
    cerr << "\n\nintErval -- " << i
         << ": valoare exterioara domeniului [ "
         << min << ", " << (max - 1) << " ].\n\n";
    exit( 1 );
  }
  return i;
}

int intErval::atr( int i ) {
  return v = verDom( i );
  // verDom(), fiind membru ca si v, se va invoca pentru
  // obiectul invocator al functiei atr()
}
```

Din cele trei inconveniente menționate în finalul Secțiunii 2.3.1 am rezolvat, până în acest moment, doar inconvenientul I_2, cel care se referă la încapsularea datelor. În continuare ne vom ocupa de I_3, adică de simplificarea sintaxei.

Limbajul C++ permite nu numai supraîncărcarea funcțiilor, ci și a majorității operatorilor predefiniți. În general, sunt posibile două modalități de supraîncărcare:

- Ca funcții membre, caz în care operandul stâng este implicit obiect invocator.
- Ca funcții nemembre, dar cu condiția ca cel puțin un argument (operand) să fie de tip clasă.

Pentru clasa `intErval`, ne interesează în primul rând operatorul de atribuire (implementat deocamdată prin funcția `atr()`) și un operator care să corespundă funcției `val()`. Deși pare surprinzător, funcția `val()` nu face altceva decât să convertească tipul `intErval` la tipul `int`. În consecință, vom implementa această funcție ca operator de conversie la `int`. În noua sa formă, clasa `intErval` arată astfel:

```
class intErval {
public:
  // operatorul de atribuire corespunzator functiei atr()
  int operator =( int i ) { return v = verDom( i ); }

  // operatorul de conversie corespunzator functiei val()
  operator int( ) { return v; }

private:
  int verDom( int );

  int min, max;
  int v;
};
```

Revenind la obiectele `indice` și `numar`, putem scrie acum

```
indice = (int)numar + 1;
```

sau direct

```
indice = numar + 1;
```

conversia `numar`-ului la `int` fiind invocată automat de către compilator. Nu este nimic miraculos în această invocare "automată", deoarece operatorul `+` nu este definit pentru argumente de tip `intErval` și `int`, dar este definit pentru `int` și `int`. Altfel spus, expresia `numar + 1` poate fi evaluată printr-o simplă conversie a primului operand de la `intErval` la `int`.

O altă funcție utilă tipului `intErval` este cea de citire a valorii `v`, funcție denumită în paragraful precedent `cit()`. Ne propunem să o înlocuim cu operatorul de extragere `>>`, pentru a putea scrie direct `cin >> numar`. Supraîncărcarea operatorului `>>` ca funcție membră nu este posibilă, deoarece argumentul stâng este obiectul invocator și atunci ar trebui să scriem `n >> cin`.

Operatorul de extragere necesar pentru citirea valorii obiectelor de tip `intErval` se poate defini astfel:

```
istream& operator >>( istream& is, intErval& n ) {
  int i;
  if ( is >> i )  // se citeste valoarea
    n = i;        // se invoca operatorul de atribuire
  return is;
}
```

Sunt două întrebări la care trebuie să răspundem referitor la funcția de mai sus:

- Care este semnificația testului `if ( is >> i )`?
- De ce se returnează `istream`-ul?

În testul `if ( is >> i )` se invocă de fapt operatorul de conversie de la `istream` la `int`, rezultatul fiind valoarea logică *true* (valoare diferită de zero) sau *false* (valoarea zero), după cum operația a decurs normal sau nu.

Returnarea `istream`-ului este o modalitate de a aplica operatorului `>>` sintaxa de concatenare, sintaxă utilizată în expresii de forma `i = j = 0`. De exemplu, obiectele `numar` și `indice` de tip `intErval`, pot fi citite printr-o singură instrucțiune

```
cin >> numar >> indice;
```

De asemenea, remarcăm și utilizarea absolut justificată a argumentelor de tip referință. În lipsa lor, obiectul `numar` ar fi putut să fie modificat doar dacă i-am fi transmis adresa. În plus, utilizarea sintaxei de concatenare provoacă, în lipsa referințelor, multiplicarea argumentului de tip `istream` de două ori pentru fiecare apel: prima dată ca argument efectiv, iar a doua oară ca valoare returnată.

Clasa `intErval` a devenit o clasă comod de utilizat, foarte bine încapsulată și cu un comportament similar întregilor. Încapsularea este însă atât de bună, încât, practic, nu avem nici o modalitate de a inițializa limitele superioară și inferioară ale domeniului admisibil. De fapt, am revenit la inconvenientul I_1 menționat în finalul Secțiunii 2.3.1. Problema inițializării datelor membre în momentul definirii obiectelor nu este specifică doar clasei `intErval`. Pentru rezolvarea ei, limbajul C++ oferă o categorie specială de funcții membre, numite *constructori*. Constructorii nu au tip, au numele identic cu numele clasei și sunt invocați automat de către compilator, după rezervarea spațiului pentru datele obiectului definit.

Constructorul necesar clasei `intErval` are ca argumente limitele domeniului admisibil. Transmiterea lor se poate face implicit, prin notația

```
intErval numar( 80, 32 );
```

sau explicit, prin specificarea constructorului

```
intErval numar = intErval( 80, 32 );
```

Definiția acestui constructor este

```
intErval::intErval( int sup, int inf ) {
  if ( inf >= sup ) {
    cerr << "\n\nintErval -- domeniu incorect specificat [ "
         << inf << ", " << (sup - 1) << " ].\n\n";
    exit( 1 );
  }
  min = v = inf;
  max =     sup;
}
```

Datorită lipsei unui constructor fără argumente, compilatorul va interzice orice declarații în care nu se specifică domeniul. De exemplu,

```
intErval indice;
```

este o definiție incompletă, semnalată la compilare. Mai mult, definițiile incorecte semantic cum este

```
intErval limita( 32, 80 );
```

sunt și ele detectate, dar nu de către compilator, ci de către constructor. Acesta, după cum se observă, verifică dacă limita inferioară a domeniului este mai mică decât cea superioară, semnalând corespunzător domeniile incorect specificate.

În declarațiile funcțiilor, limbajul C++ permite specificarea valorilor implicite ale argumentelor, valori utilizabile în situațiile în care nu se specifică toți parametrii efectivi. Această facilitate este utilă și în cazul constructorului clasei `intErval`. Prin declarația

```
intErval( int = 1, int = 0 );
```

definiția

```
intErval indice;
```

nu va mai fi respinsă, ci va provoca invocarea constructorului cu argumentele implicite `1` și `0`. Corespondența dintre argumentele actuale și cele formale se realizează pozițional, ceea ce înseamnă că primul argument este asociat limitei superioare, iar cel de-al doilea celei inferioare. Frecvent, limita inferioară are valoarea implicită zero. Deci la transmiterea argumentelor constructorului, ne putem limita doar la precizarea limitei superioare.

Constructorul apelabil fără nici un argument se numește *constructor implicit*. Altfel spus, constructorul implicit este constructorul care, fie nu are argumente, fie are toate argumentele implicite. Limbajul C++ nu impune prezența unui constructor implicit în fiecare clasă, dar sunt anumite situații în care acest constructor este absolut necesar.

După aceste ultime precizări, definiția clasei `intErval` este:

```
class intErval {
public:
  intErval( int = 1, int = 0 );
  ~intErval( ) { }

  int operator =( int i ) { return v = verDom( i ); }
  operator int( ) { return v; }

private:
  int verDom( int );

  int min, max;
  int v;
};
```

Se observă apariția unei noi funcții membre, numită `~intErval()`, al cărui corp este vid. Ea se numește *destructor*, nu are tip și nici argumente, iar numele ei este obținut prin precedarea numelui clasei de caracterul `~`. Rolul destructorului este opus celui al constructorului, în sensul că realizează operațiile necesare distrugerii corecte a obiectului. Destructorul este invocat automat, înainte de a elibera spațiul alocat datelor membre ale obiectului care încetează să mai existe. Un obiect încetează să mai existe în următoarele situații:

- Obiectele definite într-o funcție sau bloc de instrucțiuni (obiecte cu *existență locală*) încetează să mai existe la terminarea executării funcției sau blocului respectiv.
- Obiectele definite global, în exteriorul oricărei funcții, sau cele definite `static` (obiecte cu *existență statică*) încetează să mai existe la terminarea programului.
- Obiectele alocate dinamic prin operatorul `new` (obiecte cu *existență dinamică*) încetează să mai existe la invocarea operatorului `delete`.

Ca și în cazul constructorilor, prezența destructorului într-o clasă este opțională, fiind lăsată la latitudinea proiectantului clasei.

Pentru a putea fi inclusă în toate fișierele sursă în care este utilizată, definiția unei clase se introduce într-un fișier header (prefix). În scopul evitării includerii de mai multe ori a aceluiași fișier (*includeri multiple*), se recomandă ca fișierele header să aibă structura

```
#ifndef simbol
#define simbol

// continutul fisierului

#endif
```

unde `simbol` este un identificator unic în program. Dacă fișierul a fost deja inclus, atunci identificatorul `simbol` este deja definit, și deci, toate liniile situate între `#ifndef` și `#endif` vor fi ignorate. De exemplu, în fișierul `intErval.h`, care conține definiția clasei `intErval`, identificatorul `simbol` ar putea fi `__INTeRVAL_H`. Iată conținutul acestui fișier:

```
#ifndef __INTeRVAL_H
#define __INTeRVAL_H

#include <iostream.h>

class intErval {
public:
  intErval( int = 1, int = 0 );
  ~intErval( ) { }

  int operator =( int i ) { return v = verDom( i ); }
  operator int( ) { return v; }

private:
  int verDom( int );

  int min, max;
  int v;
};

istream& operator >>( istream&, intErval& );

#endif
```

Funcțiile membre se introduc într-un fișier sursă obișnuit, care este legat după compilare de programul executabil. Pentru clasa `intErval`, acest fișier este:

```
#include "intErval.h"
#include <stdlib.h>

intErval::intErval( int sup, int inf ) {
  if ( inf >= sup ) {
    cerr << "\n\nintErval -- domeniu incorect specificat [ "
         << inf << ", " << (sup - 1) << " ].\n\n";
```

```
    exit( 1 );
  }
  min = v = inf;
  max =     sup;
}

int intErval::verDom( int i ) {
  if ( i < min || i >= max ) {
    cerr << "\n\nintErval -- "
         << i << ": valoare exterioara domeniului [ "
         << min << ", " << (max - 1) << " ].\n\n";
    exit( 1 );
  }
  return i;
}

istream& operator >>( istream& is, intErval& n ) {
  int i;
  if ( is >> i )  // se citeste valoarea
    n = i;        // se invoca operatorul de atribuire
  return is;
}
```

Adaptarea programului pentru determinarea termenilor şirului lui Fibonacci necesită doar includerea fişierului `intErval.h`, precum şi schimbarea definiției rangului `n` din `int` în `intErval`.

```
#include <iostream.h>
#include "intErval.h"

long fib2( int n ) {
  long i = 1, j = 0;

  for ( int k = 0; k++ < n; j = i + j, i = j - i );
  return j;
}

int main( ) {
  cout << "\nTermenul sirului lui Fibonacci de rang ... ";

  intErval n = 47; cin >> n;
  cout << " este " << fib2( n );
  cout << '\n';

  return 0;
}
```

Desigur că, la programul executabil, se va lega şi fişierul rezultat în urma compilării definițiilor funcțiilor membre din clasa `intErval`.

Neconcordanța dintre argumentul formal de tip `int` din `fib2()` și argumentul efectiv (actual) de tip `intErval` se rezolvă, de către compilator, prin invocarea operatorului de conversie de la `intErval` la `int`.

Programarea orientată pe obiect este deosebit de avantajoasă în cazul aplicațiilor mari, dezvoltate de echipe întregi de programatori pe parcursul câtorva luni, sau chiar ani. Aplicația prezentată aici este mult prea mică pentru a putea fi folosită ca un argument în favoarea acestei tehnici de programare. Cu toate acestea, comparând cele două implementări ale clasei `intErval` (în limbajele C, respectiv C++), sunt deja evidente două avantaje ale programării orientate pe obiect:

- În primul rând, este posibilitatea dezvoltării unor tipuri noi, definite exclusiv prin comportament și nu prin structură. Codul sursă este mai compact, dar în nici un caz mai rapid decât în situația în care nu am fi folosit obiecte. Să reținem că programarea orientată pe obiect nu este o modalitate de a micșora timpul de execuție, ci de a spori eficiența activității de programare.
- În al doilea rând, se remarcă posibilitățile de a supraîncărca operatori, inclusiv pe cei de conversie. Efectul este foarte spectaculos, deoarece utilizarea noilor tipuri este la fel de comodă ca și utilizarea tipurilor predefinite. Pentru tipul `intErval`, aceste avantaje se concretizează în faptul că obiectele de tip `intErval` se comportă exact ca și cele de tip `int`, încadrarea lor în limitele domeniului admisibil fiind absolut garantată.

2.4 Exerciții *

2.1 Scrieți un program care determină termenul de rang *n* al șirului lui Fibonacci prin algoritmii *fib*1 și *fib*3.

2.2 Care sunt valorile maxime ale lui *n* pentru care algoritmii *fib*1, *fib*2 și *fib*3 returnează valori corecte? Cum pot fi mărite aceste valori?

Soluție: Presupunând că un `long` este reprezentat pe 4 octeți, atunci cel mai mare număr Fibonacci reprezentabil pe `long` este cel cu rangul 46. Lucrând pe `unsigned long`, se poate ajunge până la termenul de rang 47. Pentru aceste ranguri, timpii de execuție ai algoritmului *fib*1 diferă semnificativ de cei ai algoritmilor *fib*2 și *fib*3.

2.3 Introduceți în clasa `intErval` încă două date membre prin care să contorizați numărul de apeluri ale celor doi operatori definiți. Completați

* Chiar dacă nu se precizează explicit, toate implementările se vor realiza în limbajul C++.

constructorul și destructorul astfel încât să inițializeze, respectiv să afișeze, aceste valori.

2.4 Implementați testul de primalitate al lui Wilson prezentat în Secțiunea 1.4.

2.5 Scrieți un program pentru calculul recursiv al coeficienților binomiali după formula dată de triunghiul lui Pascal:

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & \text{pentru } 0 < k < n \\ 1 & \text{altfel} \end{cases}$$

Analizați avantajele și dezavantajele acestui program în raport cu programul care calculează coeficientul conform definiției:

$$\binom{n}{m} = \frac{n!}{m!(n-m)!}$$

Soluție: Utilizarea definiției pentru calculul combinărilor este o idee total neinspirată, nu numai în ceea ce privește eficiența, ci și pentru faptul că nu poate fi aplicată decât pentru valori foarte mici ale lui *n*. De exemplu, într-un `long` de 4 octeți, valoarea 13! nu mai poate fi calculată. Funcția recursivă este simplă:

```
int C( int n, int m) {
  return m == 0 ||
         m == n?   1: C( n - 1, m - 1 ) + C( n - 1, m );
}
```

dar și ineficientă, deoarece numărul apelurilor recursive este foarte mare (vezi Exercițiul 8.1). Programul complet este:

```
#include <iostream.h>

const int N = 16, M = 17;

int r[N][M];  // contorizeaza numarul de apeluri ale
              //   functiei C( int, int ) separat,
              //   pentru toate valorile argumentelor

long tr;      // numarul total de apeluri ale
              //   functiei C( int, int )
```

```
int C( int n, int m ) {
  r[n][m]++; tr++;
  return m == 0 || m == n?
    1: C( n - 1, m - 1 ) + C( n - 1, m );
}

void main( ) {
  int n, m;
  for ( n = 0; n < N; n++ )
    for ( m = 0; m < M; m++ ) r[n][m] = 0;
  tr = 0;

  cout << "\nCombinari de (maxim " << N << ") ... ";
  cin >> n;
  cout << "             luate cate ... ";
  cin >> m;
  cout << "sunt " << C( n, m ) << '\n';

  cout << "\n\nC( int, int ) a fost invocata de "
       << tr << " ori astfel:\n";
  for ( int i = 1; i <= n; i++, cout << '\n' )
    for ( int j = 0; j <= i; j++ ) {
      cout.width( 4 );
      cout << r[i][j] << ' ';
    }
}
```

Rezultatele obținute în urma rulării sunt următoarele:

```
Combinari de (maxim 16) ...12
             luate cate ...7
sunt 792


C( int, int ) a fost invocata de 1583 ori astfel:
 210  210
  84  210  126
  28   84  126   70
   7   28   56   70   35
   1    7   21   35   35   15
   0    1    6   15   20   15    5
   0    0    1    5   10   10    5    1
   0    0    0    1    4    6    4    1    0
   0    0    0    0    1    3    3    1    0    0
   0    0    0    0    0    1    2    1    0    0    0
   0    0    0    0    0    0    1    1    0    0    0    0
   0    0    0    0    0    0    0    1    0    0    0  ...
```

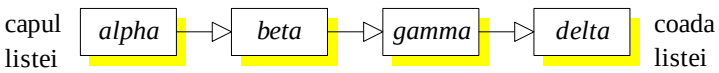
Se observă că `C(1,1)` a fost invocată de 210 ori, iar `C(2,2)` de 126 de ori!

3. Structuri elementare de date

Înainte de a elabora un algoritm, trebuie să ne gândim la modul în care reprezentăm datele. În acest capitol vom trece în revistă structurile fundamentale de date cu care vom opera. Presupunem în continuare că sunteți deja familiarizați cu noțiunile de fișier, tablou, listă, graf, arbore și ne vom concentra mai ales pe prezentarea unor concepte mai particulare: heap-uri și structuri de mulțimi disjuncte.

3.1 Liste

O *listă* este o colecție de elemente de informație (noduri) aranjate într-o anumită ordine. *Lungimea* unei liste este numărul de noduri din listă. Structura corespunzătoare de date trebuie să ne permită să determinăm eficient care este primul/ultimul nod în structură și care este predecesorul/succesorul (dacă există) unui nod dat. Iată cum arată cea mai simplă listă, *lista liniară*:



O *listă circulară* este o listă în care, după ultimul nod, urmează primul, deci fiecare nod are succesor și predecesor.

Operații curente care se fac în liste sunt: inserarea unui nod, ștergerea (extragerea) unui nod, concatenarea unor liste, numărarea elementelor unei liste etc. Implementarea unei liste se poate face în principal în două moduri:

- *Implementarea secvențială*, în locații succesive de memorie, conform ordinii nodurilor în listă. Avantajele acestei tehnici sunt accesul rapid la predecesorul/succesorul unui nod și găsirea rapidă a primului/ultimului nod. Dezavantajele sunt inserarea/ștergerea relativ complicată a unui nod și faptul că, în general, nu se folosește întreaga memorie alocată listei.
- *Implementarea înlănțuită*. În acest caz, fiecare nod conține două părți: informația propriu-zisă și adresa nodului succesor. Alocarea memoriei fiecărui nod se poate face în mod dinamic, în timpul rulării programului. Accesul la un nod necesită parcurgerea tuturor predecesorilor săi, ceea ce poate lua ceva mai

mult timp. Inserarea/ștergerea unui nod este în schimb foarte rapidă. Se pot folosi două adrese în loc de una, astfel încât un nod să conțină pe lângă adresa nodului succesor și adresa nodului predecesor. Obținem astfel o listă *dublu înlănțuită*, care poate fi traversată în ambele direcții.

Listele implementate înlănțuit pot fi reprezentate cel mai simplu prin tablouri. În acest caz, adresele sunt de fapt indici de tablou. O alternativă este să folosim tablouri paralele: să memorăm informația fiecărui nod (valoarea) într-o locație *VAL*[*i*] a tabloului *VAL*[1 .. *n*], iar adresa (indicele) nodului său succesor într-o locație *LINK*[*i*] a tabloului *LINK*[1 .. *n*]. Indicele de tablou al locației primului nod este memorat în variabila *head*. Vom conveni ca, pentru cazul listei vide, să avem *head* = 0. Convenim de asemenea ca *LINK*[ultimul nod din listă] = 0. Atunci, *VAL*[*head*] va conține informația primului nod al listei, *LINK*[*head*] adresa celui de-al doilea nod, *VAL*[*LINK*[*head*]] informația din al doilea nod, *LINK*[*LINK*[*head*]] adresa celui de-al treilea nod etc.

Acest mod de reprezentare este simplu dar, la o analiză mai atentă, apare o problemă esențială: cea a gestionării locațiilor libere. O soluție elegantă este să reprezentăm locațiile libere tot sub forma unei liste înlănțuite. Atunci, ștergerea unui nod din lista inițială implică inserarea sa în lista cu locații libere, iar inserarea unui nod în lista inițială implică ștergerea sa din lista cu locații libere. Aspectul cel mai interesant este că, pentru implementarea listei de locații libere, putem folosi aceleași tablouri. Avem nevoie de o altă variabilă, *freehead*, care va conține indicele primei locații libere din *VAL* și *LINK*. Folosim aceleași convenții: dacă *freehead* = 0 înseamnă că nu mai avem locații libere, iar *LINK*[ultima locație liberă] = 0.

Vom descrie în continuare două tipuri de liste particulare foarte des folosite.

3.1.1 Stive

O *stivă* (*stack*) este o listă liniară cu proprietatea că operațiile de inserare/extragere a nodurilor se fac în/din coada listei. Dacă nodurile A, B, C, D sunt inserate într-o stivă în această ordine, atunci primul nod care poate fi extras este D. În mod echivalent, spunem că ultimul nod inserat va fi și primul șters. Din acest motiv, stivele se mai numesc și liste *LIFO* (**L**ast **I**n **F**irst **O**ut), sau liste *pushdown*.

Cel mai natural mod de reprezentare pentru o stivă este implementarea secvențială într-un tablou *S*[1 .. *n*], unde *n* este numărul maxim de noduri. Primul nod va fi memorat în *S*[1], al doilea în *S*[2], iar ultimul în *S*[*top*], unde *top* este o variabilă care conține adresa (indicele) ultimului nod inserat. Inițial, când stiva este vidă, avem *top* = 0. Iată algoritmii de inserare și de ștergere (extragere) a unui nod:

function *push*(*x*, *S*[1 .. *n*])
{adaugă nodul *x* în stivă}
if *top* ≥ *n* **then return** "*stivă plină*"
top ⟵ *top*+1
S[*top*] ⟵ *x*
return "*succes*"

function *pop*(*S*[1 .. *n*])
{șterge ultimul nod inserat din stivă și îl returnează}
if *top* ≤ 0 **then return** "*stivă vidă*"
x ⟵ *S*[*top*]
top ⟵ *top*−1
return *x*

Cei doi algoritmi necesită timp constant, deci nu depind de mărimea stivei.

Vom da un exemplu elementar de utilizare a unei stive. Dacă avem de calculat expresia aritmetică

$$5*(((9+8)*(4*6))+7)$$

putem folosi o stivă pentru a memora rezultatele intermediare. Într-o scriere simplificată, iată cum se poate calcula expresia de mai sus:

push(5); *push*(9); *push*(8); *push*(*pop* + *pop*); *push*(4); *push*(6);
push(*pop* ∗ *pop*); *push*(*pop* ∗ *pop*); *push*(7); *push*(*pop* + *pop*);
push(*pop* ∗ *pop*); **write** (*pop*);

Observăm că, pentru a efectua o operație aritmetică, trebuie ca operanzii să fie deja în stivă atunci când întâlnim operatorul. Orice expresie aritmetică poate fi transformată astfel încât să îndeplinească această condiție. Prin această transformare se obține binecunoscuta notație postfixată (sau poloneză inversă), care se bucură de o proprietate remarcabilă: nu sunt necesare paranteze pentru a indica ordinea operațiilor. Pentru exemplul de mai sus, notația postfixată este:

5 9 8 + 4 6 * * 7 + *

3.1.2 Cozi

O *coadă* (*queue*) este o listă liniară în care inserările se fac doar în capul listei, iar extragerile doar din coada listei. Cozile se numesc și liste *FIFO* (**F**irst **I**n **F**irst **O**ut).

O reprezentare secvențială interesantă pentru o coadă se obține prin utilizarea unui tablou *C*[0 .. *n*−1], pe care îl tratăm ca și cum ar fi circular: după locația *C*[*n*−1] urmează locația *C*[0]. Fie *tail* variabila care conține indicele locației

predecesoare primei locații ocupate și fie *head* variabila care conține indicele locației ocupate ultima oară. Variabilele *head* și *tail* au aceeași valoare atunci și numai atunci când coada este vidă. Inițial, avem *head* = *tail* = 0. Inserarea și ștergerea (extragerea) unui nod necesită timp constant.

```
function insert-queue(x, C[0 .. n−1])
    {adaugă nodul x în capul cozii}
    head ← (head+1) mod n
    if head = tail then return "coadă plină"
    C[head] ← x
    return "succes"

function delete-queue(C[0 .. n−1])
    {șterge nodul din coada listei și îl returnează}
    if head = tail then return "coadă vidă"
    tail ← (tail+1) mod n
    x ← C[tail]
    return x
```

Este surprinzător faptul că testul de coadă vidă este același cu testul de coadă plină. Dacă am folosi toate cele *n* locații, atunci nu am putea distinge între situația de "*coadă plină*" și cea de "*coadă vidă*", deoarece în ambele situații am avea *head* = *tail*. În consecință, se folosesc efectiv numai $n-1$ locații din cele *n* ale tabloului *C*, deci se pot implementa astfel cozi cu cel mult $n-1$ noduri.

3.2 Grafuri

Un *graf* este o pereche $G = \langle V, M \rangle$, unde V este o mulțime de vârfuri, iar $M \subseteq V \times V$ este o mulțime de muchii. O muchie de la vârful a la vârful b este notată cu perechea ordonată (a, b), dacă graful este *orientat*, și cu mulțimea $\{a, b\}$, dacă graful este *neorientat*. În cele ce urmează vom presupune că vârfurile a și b sunt diferite. Două vârfuri unite printr-o muchie se numesc *adiacente*. Un drum este o succesiune de muchii de forma

$$(a_1, a_2), (a_2, a_3), \ldots, (a_{n-1}, a_n)$$

sau de forma

$$\{a_1, a_2\}, \{a_2, a_3\}, \ldots, \{a_{n-1}, a_n\}$$

după cum graful este orientat sau neorientat. *Lungimea* drumului este egală cu numărul muchiilor care îl constituie. Un *drum simplu* este un drum în care nici un vârf nu se repetă. Un *ciclu* este un drum care este simplu, cu excepția primului și ultimului vârf, care coincid. Un *graf aciclic* este un graf fără cicluri. Un *subgraf* al lui G este un graf $\langle V', M' \rangle$, unde $V' \subseteq V$, iar M' este formată din muchiile din M care unesc vârfuri din V'. Un *graf parțial* este un graf $\langle V, M'' \rangle$, unde $M'' \subseteq M$.

Un graf neorientat este *conex*, dacă între oricare două vârfuri există un drum. Pentru grafuri orientate, această noțiune este întărită: un graf orientat este *tare conex*, dacă între oricare două vârfuri i și j există un drum de la i la j și un drum de la j la i.

În cazul unui graf neconex, se pune problema determinării componentelor sale conexe. O *componentă conexă* este un subgraf conex maximal, adică un subgraf conex în care nici un vârf din subgraf nu este unit cu unul din afară printr-o muchie a grafului inițial. Împărțirea unui graf $G = <V, M>$ în componentele sale conexe determină o partiție a lui V și una a lui M.

Un *arbore* este un graf neorientat aciclic conex. Sau, echivalent, un arbore este un graf neorientat în care există exact un drum între oricare două vârfuri[*]. Un graf parțial care este arbore se numește *arbore parțial*.

Vârfurilor unui graf li se pot atașa informații numite uneori *valori*, iar muchiilor li se pot atașa informații numite uneori *lungimi* sau *costuri*.

Există cel puțin trei moduri evidente de reprezentare ale unui graf:

- Printr-o *matrice de adiacență* A, în care $A[i, j] = true$ dacă vârfurile i și j sunt adiacente, iar $A[i, j] = false$ în caz contrar. O variantă alternativă este să-i dăm lui $A[i, j]$ valoarea lungimii muchiei dintre vârfurile i și j, considerând $A[i, j] = +\infty$ atunci când cele două vârfuri nu sunt adiacente. Memoria necesară este în ordinul lui n^2. Cu această reprezentare, putem verifica ușor dacă două vârfuri sunt adiacente. Pe de altă parte, dacă dorim să aflăm toate vârfurile adiacente unui vârf dat, trebuie să analizăm o întreagă linie din matrice. Aceasta necesită n operații (unde n este numărul de vârfuri în graf), independent de numărul de muchii care conectează vârful respectiv.
- Prin *liste de adiacență*, adică prin atașarea la fiecare vârf i a listei de vârfuri adiacente lui (pentru grafuri orientate, este necesar ca muchia să plece din i). Într-un graf cu m muchii, suma lungimilor listelor de adiacență este $2m$, dacă graful este neorientat, respectiv m, dacă graful este orientat. Dacă numărul muchiilor în graf este mic, această reprezentare este preferabilă din punct de vedere al memoriei necesare. Este posibil să examinăm toți vecinii unui vârf dat, în medie, în mai puțin de n operații. Pe de altă parte, pentru a determina dacă două vârfuri i și j sunt adiacente, trebuie să analizăm lista de adiacență a lui i (și, posibil, lista de adiacență a lui j), ceea ce este mai puțin eficient decât consultarea unei valori logice în matricea de adiacență.
- Printr-o *listă de muchii*. Această reprezentare este eficientă atunci când avem de examinat toate muchiile grafului.

[*] În Exercițiul 3.2 sunt date și alte propoziții echivalente care caracterizează un arbore.

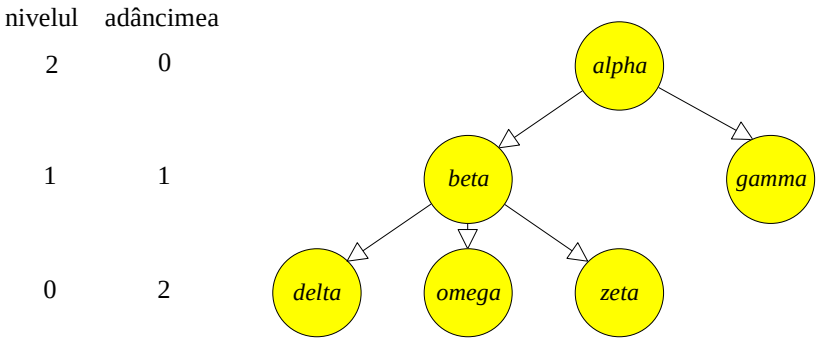


Figura 3.1 Un arbore cu rădăcină.

3.3 Arbori cu rădăcină

Fie *G* un graf orientat. *G* este un *arbore cu rădăcina r*, dacă există în *G* un vârf *r* din care oricare alt vârf poate fi ajuns printr-un drum unic.

Definiția este valabilă și pentru cazul unui graf neorientat, alegerea unei rădăcini fiind însă în acest caz arbitrară: orice arbore este un arbore cu rădăcină, iar rădăcina poate fi fixată în oricare vârf al său. Aceasta, deoarece dintr-un vârf oarecare se poate ajunge în oricare alt vârf printr-un drum unic.

Când nu va fi pericol de confuzie, vom folosi termenul “arbore”, în loc de termenul corect “arbore cu rădăcină”. Cel mai intuitiv este să reprezentăm un arbore cu rădăcină, ca pe un arbore propriu-zis. În Figura 3.1, vom spune că *beta* este *tatăl* lui *delta* și *fiul* lui *alpha*, că *beta* și *gamma* sunt *frați*, că *delta* este un *descendent* al lui *alpha*, iar *alpha* este un *ascendent* al lui *delta*. Un vârf *terminal* este un vârf fără descendenți. Vârfurile care nu sunt terminale sunt *neterminale*. De multe ori, vom considera că există o ordonare a descendenților aceluiași părinte: *beta* este situat la stânga lui *gamma*, adică *beta* este fratele mai vârstnic al lui *gamma*.

Orice vârf al unui arbore cu rădăcină este rădăcina unui *subarbore* constând din vârful respectiv și toți descendenții săi. O mulțime de arbori disjuncți formează o *pădure*.

Într-un arbore cu rădăcină vom adopta următoarele notații. *Adâncimea* unui vârf este lungimea drumului dintre rădăcină și acest vârf; *înălțimea* unui vârf este lungimea celui mai lung drum dintre acest vârf și un vârf terminal; *înălțimea*

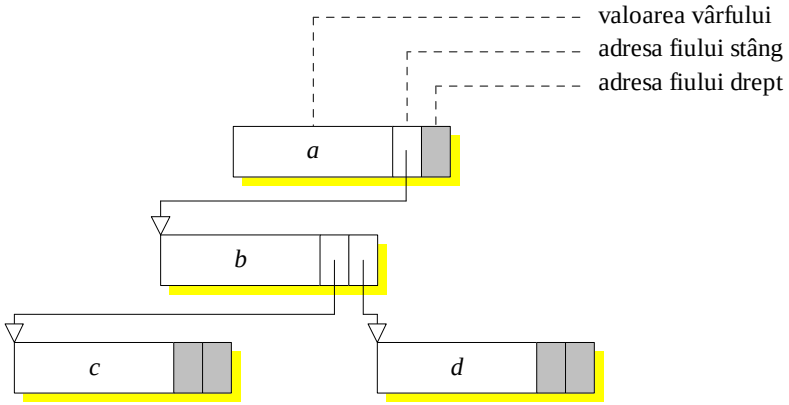


Figura 3.2 Reprezentarea prin adrese a unui arbore binar.

arborelui este înălțimea rădăcinii; *nivelul* unui vârf este înălțimea arborelui, minus adâncimea acestui vârf.

Reprezentarea unui arbore cu rădăcină se poate face prin adrese, ca și în cazul listelor înlănțuite. Fiecare vârf va fi memorat în trei locații diferite, reprezentând informația propriu-zisă a vârfului (valoarea vârfului), adresa celui mai vârstnic fiu și adresa următorului frate. Păstrând analogia cu listele înlănțuite, dacă se cunoaște de la început numărul maxim de vârfuri, atunci implementarea arborilor cu rădăcină se poate face prin tablouri paralele.

Dacă fiecare vârf al unui arbore cu rădăcină are până la n fii, arborele respectiv este n-ar. Un arbore binar poate fi reprezentat prin adrese, ca în Figura 3.2. Observăm că pozițiile pe care le ocupă cei doi fii ai unui vârf sunt semnificative: lui a îi lipsește *fiul drept*, iar b este *fiul stâng* al lui a.

Într-un arbore binar, numărul maxim de vârfuri de adâncime k este 2^k. Un arbore binar de înălțime i are cel mult $2^{i+1}-1$ vârfuri, iar dacă are exact $2^{i+1}-1$ vârfuri, se numește *arbore plin*. Vârfurile unui arbore plin se numerotează în ordinea adâncimii. Pentru aceeași adâncime, numerotarea se face în arbore de la stânga la dreapta (Figura 3.3).

Un arbore binar cu n vârfuri și de înălțime i este *complet*, dacă se obține din arborele binar plin de înălțime i, prin eliminarea, dacă este cazul, a vârfurilor numerotate cu $n+1, n+2, \ldots, 2^{i+1}-1$. Acest tip de arbore se poate reprezenta secvențial folosind un tablou T, punând vârfurile de adâncime k, de la stânga la dreapta, în pozițiile $T[2^k], T[2^{k+1}], \ldots, T[2^{k+1}-1]$ (cu posibila excepție a nivelului 0, care poate fi incomplet). De exemplu, Figura 3.4 exemplifică cum poate fi

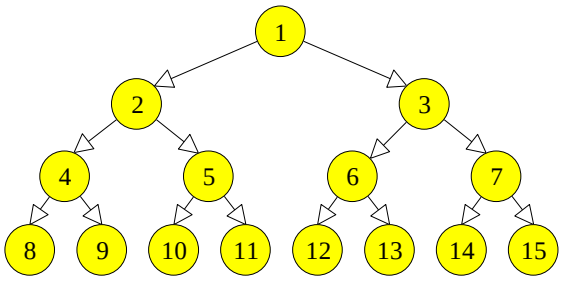


Figura 3.3 Numerotarea vârfurilor într-un arbore binar de înălțime 3.

reprezentat un arbore binar complet cu zece vârfuri, obținut din arborele plin din Figura 3.3, prin eliminarea vârfurilor 11, 12, 13, 14 și 15. Tatăl unui vârf reprezentat în $T[i]$, $i > 1$, se află în $T[i \textbf{ div } 2]$. Fiii unui vârf reprezentat în $T[i]$ se află, dacă există, în $T[2i]$ și $T[2i+1]$.

Facem acum o scurtă incursiune în matematica elementară, pentru a stabili câteva rezultate de care vom avea nevoie în capitolele următoare. Pentru un număr real oarecare x, definim

$$\lfloor x \rfloor = \max\{n \mid n \le x, n \text{ este întreg}\} \quad \text{și} \quad \lceil x \rceil = \min\{n \mid n \ge x, n \text{ este întreg}\}$$

Puteți demonstra cu ușurință următoarele proprietăți:

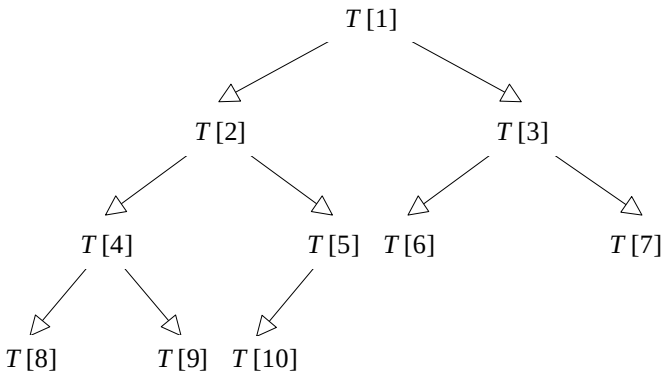


Figura 3.4 Un arbore binar complet.

i) $x-1 < \lfloor x \rfloor \le x \le \lceil x \rceil < x+1$
pentru orice x real

ii) $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$
pentru orice n întreg

iii) $\lceil \lceil n/a \rceil / b \rceil = \lceil n/ab \rceil$ și $\lfloor \lfloor n/a \rfloor / b \rfloor = \lfloor n/ab \rfloor$
pentru orice n, a, b întregi (a, $b \neq 0$)

iv) $\lfloor n/m \rfloor = \lceil (n-m+1)/m \rceil$ și $\lceil n/m \rceil = \lfloor (n+m-1)/m \rfloor$
pentru orice numere întregi pozitive n și m

În fine, arătați că un arbore binar complet cu n vârfuri are înălțimea $\lfloor \lg n \rfloor$.

3.4 Heap-uri

Un *heap* (în traducere aproximativă, "grămadă ordonată") este un arbore binar complet, cu următoarea proprietate, numită *proprietate de heap*: valoarea fiecărui vârf este mai mare sau egală cu valoarea fiecărui fiu al său. Figura 3.5 prezintă un exemplu de heap.

Același heap poate fi reprezentat secvențial prin următorul tablou:

10	7	9	4	7	5	2	2	1	6
$T[1]$	$T[2]$	$T[3]$	$T[4]$	$T[5]$	$T[6]$	$T[7]$	$T[8]$	$T[9]$	$T[10]$

Caracteristica de bază a acestei structuri de dată este că modificarea valorii unui vârf se face foarte eficient, păstrându-se proprietatea de heap. Dacă valoarea unui vârf crește, astfel încât depășește valoarea tatălui, este suficient să schimbăm între ele aceste două valori și să continuăm procedeul în mod ascendent, până când proprietatea de heap este restabilită. Vom spune că valoarea modificată a fost *filtrată* (*percolated*) către noua sa poziție. Dacă, dimpotrivă, valoarea vârfului

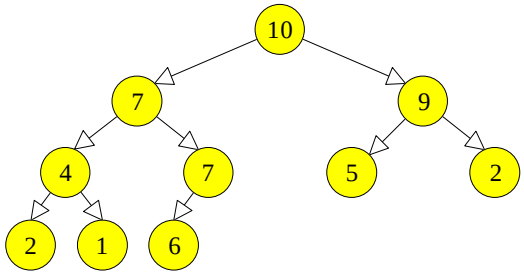


Figura 3.5 Un heap.

scade, astfel încât devine mai mică decât valoarea cel puțin a unui fiu, este suficient să schimbăm între ele valoarea modificată cu cea mai mare valoare a fiiilor, apoi să continuăm procesul în mod descendent, până când proprietatea de heap este restabilită. Vom spune că valoarea modificată a fost *cernută* (*sifted down*) către noua sa poziție. Următoarele proceduri descriu formal operațiunea de modificare a valorii unui vârf într-un heap.

```
procedure alter-heap(T[1 .. n], i, v)
    {T[1 .. n] este un heap; lui T[i], 1 ≤ i ≤ n, i se atribuie
      valoarea v și proprietatea de heap este restabilită}
    x ← T[i]
    T[i] ← v
    if v < x   then  sift-down(T, i)
               else  percolate(T, i)

procedure sift-down(T[1 .. n], i)
    {se cerne valoarea din T[i]}
    k ← i
    repeat
        j ← k
        {găsește fiul cu valoarea cea mai mare}
        if 2j ≤ n and T[2j] > T[k]   then k ← 2j
        if 2j < n and T[2j+1] > T[k]   then k ← 2j+1
        interschimbă T[ j] și T[k]
    until j = k

procedure percolate(T[1 .. n], i)
    {se filtrează valoarea din T[i]}
    k ← i
    repeat
        j ← k
        if j > 1 and T[ j div 2] < T[k] then k ← j div 2
        interschimbăT[ j] și T[k]
    until j = k
```

Heap-ul este structura de date ideală pentru determinarea și extragerea maximului dintr-o mulțime, pentru inserarea unui vârf, pentru modificarea valorii unui vârf. Sunt exact operațiile de care avem nevoie pentru a implementa o listă dinamică de priorități: valoarea unui vârf va da prioritatea evenimentului corespunzător. Evenimentul cu prioritatea cea mai mare se va afla mereu la rădăcina heap-ului, iar prioritatea unui eveniment poate fi modificată în mod dinamic. Algoritmii care efectuează aceste operații sunt:

```
function find-max(T[1 .. n])
    {returnează elementul cel mai mare din heap-ul T}
    return T[1]
```

```
procedure delete-max(T[1 .. n])
    {șterge elementul cel mai mare din heap-ul T}
    T[1] ← T[n]
    sift-down(T[1 .. n−1], 1)

procedure insert(T[1 .. n], v)
    {inserează un element cu valoarea v în heap-ul T
     și restabilește proprietatea de heap}
    T[n+1] ← v
    percolate(T[1 .. n+1], n+1)
```

Rămâne de văzut cum putem forma un heap pornind de la tabloul neordonat $T[1 .. n]$. O soluție evidentă este de a porni cu un heap vid și să adăugăm elementele unul câte unul.

```
procedure slow-make-heap(T[1 .. n])
    {formează, în mod ineficient, din T un heap}
    for i ← 2 to n do percolate(T[1 .. i], i)
```

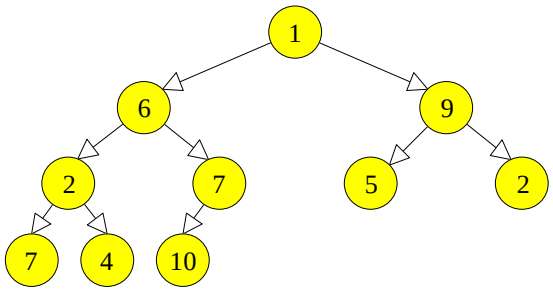
Soluția nu este eficientă și, în Capitolul 5, vom reveni asupra acestui lucru. Există din fericire un algoritm mai inteligent, care lucrează în timp liniar, după cum vom demonstra tot în Capitolul 5.

```
procedure make-heap(T[1 .. n])
    {formează din T un heap}
    for i ← (n div 2) downto 1 do sift-down[T, i]
```

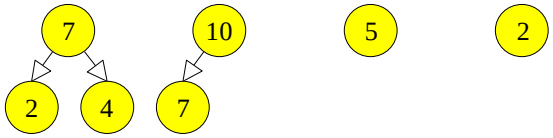
Ne reamintim că în $T[n \textbf{ div } 2]$ se află tatăl vârfului din $T[n]$. Pentru a înțelege cum lucrează această procedură, să presupunem că pornim de la tabloul:

1	6	9	2	7	5	2	7	4	10

care corespunde arborelui:



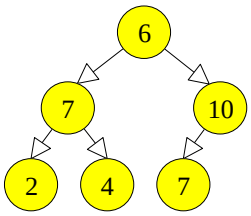
Mai întâi formăm heap-uri din subarborii cu rădăcina la nivelul 1, aplicând procedura *sift-down* rădăcinilor respective:



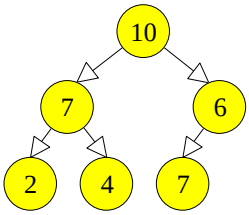
După acest pas, tabloul *T* devine:

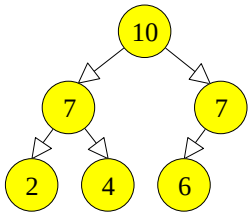
1	6	9	7	10	5	2	2	4	7

Subarborii de la următorul nivel sunt apoi transformați și ei în heap-uri. Astfel, subarborele



se transformă succesiv în:





Subarborele de nivel 2 din dreapta este deja heap. După acest pas, tabloul *T* devine:

1	10	9	7	7	5	2	2	4	6

Urmează apoi să repetăm procedeul și pentru nivelul 3, obținând în final heap-ul din Figura 3.5.

Un *min-heap* este un heap în care proprietatea de heap este inversată: valoarea fiecărui vârf este mai mică sau egală cu valoarea fiecărui fiu al său. Evident, rădăcina unui min-heap va conține în acest caz cel mai mic element al heap-ului. În mod corespunzător, se modifică și celelalte proceduri de manipulare a heap-ului.

Chiar dacă heap-ul este o structură de date foarte atractivă, există totuși și operații care nu pot fi efectuate eficient într-un heap. O astfel de operație este, de exemplu, găsirea unui vârf având o anumită valoare dată.

Conceptul de heap poate fi îmbunătățit în mai multe feluri. Astfel, pentru aplicații în care se folosește mai des procedura *percolate* decât procedura *sift-down*, rentează ca un vârf neterminal să aibă mai mult de doi fii. Aceasta accelerează procedura *percolate*. Și un astfel de heap poate fi implementat secvențial.

Heap-ul este o structură de date cu numeroase aplicații, inclusiv o remarcabilă tehnică de sortare, numită *heapsort*.

```
procedure heapsort(T[1 .. n])
    {sortează tabloul T}
    make-heap(T)
    for i ← n downto 2 do
        interschimbă T[1] și T[i]
        sift-down(T[1 .. i−1], 1)
```

Structura de heap a fost introdusă (Williams, 1964) tocmai ca instrument pentru acest algoritm de sortare.

3.5 Structuri de mulțimi disjuncte

Să presupunem că avem N elemente, numerotate de la 1 la N. Numerele care identifică elementele pot fi, de exemplu, indici într-un tablou unde sunt memorate numele elementelor. Fie o partiție a acestor N elemente, formată din submulțimi două câte două disjuncte: S_1, S_2, … . Ne interesează să rezolvăm două probleme:

i) Cum să obținem reuniunea a două submulțimi, $S_i \cup S_j$.

ii) Cum să găsim submulțimea care conține un element dat.

Avem nevoie de o structură de date care să permită rezolvarea eficientă a acestor probleme.

Deoarece submulțimile sunt două câte două disjuncte, putem alege ca etichetă pentru o submulțime oricare element al ei. Vom conveni pentru început ca elementul minim al unei mulțimi să fie eticheta mulțimii respective. Astfel, mulțimea {3, 5, 2, 8} va fi numită “mulțimea 2”.

Vom aloca tabloul *set*[1 .. N], în care fiecărei locații *set*[i] i se atribuie eticheta submulțimii care conține elementul i. Avem atunci proprietatea: *set*[i] ≤ i, pentru $1 \le i \le N$.

Presupunem că, inițial, fiecare element formează o submulțime, adică *set*[i] = i, pentru $1 \le i \le N$. Problemele *i)* și *ii)* se pot rezolva prin următorii algoritmi:

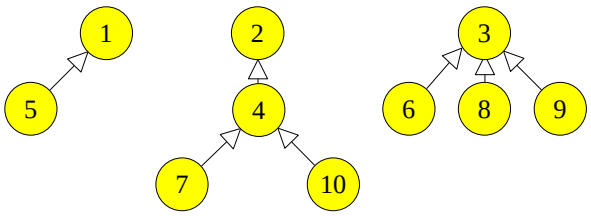
```
function find1(x)
    {returnează eticheta mulțimii care îl conține pe x}
    return set[x]

procedure merge1(a, b)
    {fuzionează mulțimile etichetate cu a și b}
    i ← a; j ← b
    if i > j then interschimbă i și j
    for k ← j to N do
        if set[k] = j then set[k] ← i
```

Dacă consultarea sau modificarea unui element dintr-un tablou contează ca o operație elementară, atunci se poate demonstra (Exercițiul 3.7) că o serie de n operații *merge*1 și *find*1 necesită, pentru cazul cel mai nefavorabil și pornind de la starea inițială, un timp în ordinul lui n^2.

Încercăm să îmbunătățim acești algoritmi. Folosind în continuare același tablou, vom reprezenta fiecare mulțime ca un arbore cu rădăcină ”inversat”. Adoptăm următoarea tehnică: dacă $set[i] = i$, atunci i este atât eticheta unei mulțimi, cât și rădăcina arborelui corespunzător; dacă $set[i] = j \neq i$, atunci j este tatăl lui i într-un arbore. De exemplu, tabloul:

1	2	3	2	1	3	4	3	3	4
set[1]	*set*[2]	…							*set*[10]

reprezintă arborii:



care, la rândul lor, reprezintă mulțimile {1,5}, {2,4,7,10} și {3,6,8,9}. Pentru a fuziona două mulțimi, trebuie acum să modificăm doar o singură valoare în tablou; pe de altă parte, este mai dificil să găsim mulțimea căreia îi aparține un element dat.

```
function find2(x)
    {returnează eticheta mulțimii care îl conține pe x}
    i ← x
    while set[i] ≠ i do i ← set[i]
    return i
```

```
procedure merge2(a, b)
    {fuzionează mulțimile etichetate cu a și b}
    if a < b   then  set[b] ← a
               else  set[a] ← b
```

O serie de n operații *find*2 și *merge*2 necesită, pentru cazul cel mai nefavorabil și pornind de la starea inițială, un timp tot în ordinul lui n^2 (Exercițiul 3.7). Deci, deocamdată, nu am câștigat nimic față de prima variantă a acestor algoritmi. Aceasta deoarece după k apeluri ale lui *merge*2, se poate să ajungem la un arbore de înălțime k, astfel încât un apel ulterior al lui *find*2 să ne pună în situația de a parcurge k muchii până la rădăcină.

Până acum am ales (arbitrar) ca elementul minim să fie eticheta unei mulțimi. Când fuzionăm doi arbori de înălțime h_1 și respectiv h_2, este bine să facem astfel încât rădăcina arborelui de înălțime mai mică să devină fiu al celeilalte rădăcini. Atunci, înălțimea arborelui rezultat va fi $\max(h_1, h_2)$, dacă $h_1 \neq h_2$, sau h_1+1, dacă $h_1 = h_2$. Vom numi această tehnică *regulă de ponderare*. Aplicarea ei implică renunțarea la convenția ca elementul minim să fie eticheta mulțimii respective. Avantajul este că înălțimea arborilor nu mai crește atât de rapid. Putem demonstra (Exercițiul 3.9) că folosind regula de ponderare, după un număr arbitrar de fuzionări, pornind de la starea inițială, un arbore având k vârfuri va avea înălțimea maximă $\lfloor \lg k \rfloor$.

Înălțimea arborilor poate fi memorată într-un tablou $H[1 .. N]$, astfel încât $H[i]$ să conțină înălțimea vârfului i în arborele său curent. În particular, dacă a este eticheta unei mulțimi, $H[a]$ va conține înălțimea arborelui corespunzător. Inițial, $H[i] = 0$ pentru $1 \leq i \leq N$. Algoritmul *find*2 rămâne valabil, dar vom modifica algoritmul de fuzionare.

```
procedure merge3(a, b)
    {fuzionează mulțimile etichetate cu a și b;
    presupunem că a ≠ b}
    if H[a] = H[b]
       then  H[a] ← H[a]+1
             set[b] ← a
       else  if H[a] > H[b]
                then  set[b] ← a
                else  set[a] ← b
```

O serie de n operații *find*2 și *merge*3 necesită, pentru cazul cel mai nefavorabil și pornind de la starea inițială, un timp în ordinul lui $n \log n$.

Continuăm cu îmbunătățirile, modificând algoritmul *find*2. Vom folosi tehnica *comprimării drumului*, care constă în următoarele. Presupunând că avem de determinat mulțimea care îl conține pe x, traversăm (conform cu *find*2) muchiile care conduc spre rădăcina arborelui. Cunoscând rădăcina, traversăm aceleași

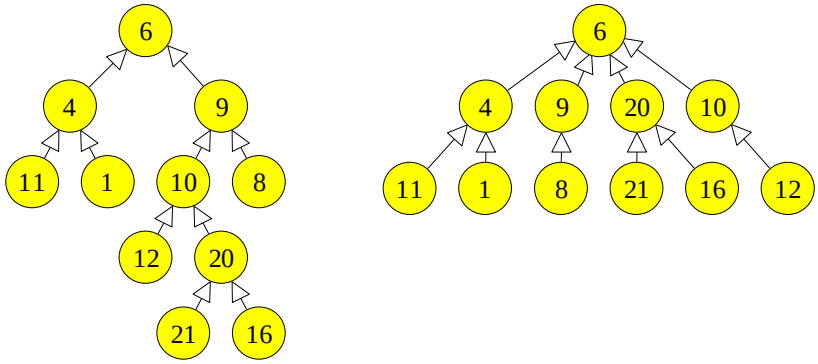


Figura 3.6 Comprimarea drumului.

muchii din nou, modificând acum fiecare vârf întâlnit în cale astfel încât să conțină direct adresa rădăcinii. Folosind tehnica comprimării drumului, nu mai este adevărat că înălțimea unui arbore cu rădăcina a este dată de $H[a]$. Totuși, $H[a]$ reprezintă în acest caz o limită superioară a înălțimii și procedura *merge*3 rămâne, cu această observație, valabilă. Algoritmul *find*2 devine:

```
function find3(x)
    {returnează eticheta mulțimii care îl conține pe x}
    r ← x
    while set[r] ≠ r do r ← set[r]
    {r este rădăcina arborelui}
    i ← x
    while i ≠ r do
        j ← set[i]
        set[i] ← r
        i ← j
    return r
```

De exemplu, executând operația *find*3(20) asupra arborelui din Figura 3.6a, obținem arborele din Figura 3.6b.

Algoritmii *find*3 și *merge*3 sunt o variantă considerabil îmbunătățită a procedurilor de tip *find* și *merge*. O serie de n operații *find*3 și *merge*3 necesită, pentru cazul cel mai nefavorabil și pornind de la starea inițială, un timp în ordinul lui $n \lg^* N$, unde $\lg^*$ este definit astfel:

$$\lg^* N = \min\{k \mid \underbrace{\lg \lg \ldots \lg}_{\text{de } k \text{ ori}} N \leq 0\}$$

Demonstrarea acestei afirmații este laborioasă și nu o vom prezenta aici. Funcția $\lg^*$ crește extrem de încet: $\lg^* N \leq 5$ pentru orice $N \leq 65536$ și $\lg^* N \leq 6$ pentru orice $N \leq 2^{65536}$. Deoarece numărul atomilor universului observabil este estimat la aproximativ 10^{80}, ceea ce este mult mai puțin decât 2^{65536}, vom întâlni foarte rar o valoare a lui N pentru care $\lg^* N > 6$.

De acum încolo, atunci când vom aplica procedurile *find*3 și *merge*3 asupra unor mulțimi disjuncte de elemente, vom spune că folosim o *structură de mulțimi disjuncte*.

O importantă aplicație practică a structurilor de mulțimi disjuncte este verificarea eficientă a conexității unui graf (Exercițiul 3.12).

3.6 Exerciții

3.1 Scrieți algoritmii de inserare și de ștergere a unui nod pentru o stivă implementată prin tehnica tablourilor paralele.

3.2 Fie G un graf neorientat cu n vârfuri, $n \geq 2$. Demonstrați echivalența următoarelor propoziții care caracterizează un arbore:

i) G este conex și aciclic.

ii) G este aciclic și are $n-1$ muchii.

iii) G este conex și are $n-1$ muchii.

iv) G este aciclic și, adăugându-se o singură muchie între oricare două vârfuri neadiacente, se crează exact un ciclu.

v) G este conex și, dacă se suprimă o muchie oarecare, nu mai este conex.

vi) Oricare două vârfuri din G sunt unite printr-un drum unic.

3.3 Elaborați și implementați un algoritm de evaluare a expresiilor aritmetice postfixate.

3.4 De ce procedura *percolate* este mai eficientă dacă admitem că un vârf neterminal poate avea mai mult de doi fii?

3.5 Fie $T[1 .. 12]$ un tablou, astfel încât $T[i] = i$, pentru $i < 12$. Determinați starea tabloului după fiecare din următoarele apeluri de procedură, aplicate succesiv:

make-heap(T); *alter-heap*(T, 12, 10); *alter-heap*(T, 1, 6); *alter-heap*(T, 5, 6)

3.6 Implementați un model de simulare a unei liste dinamice de priorități folosind structura de heap.

3.7 În situația în care, consultarea sau modificarea unui element din tablou contează ca o operație elementară, demonstrați că timpul de execuție necesar pentru o secvență de n operații *find*1 și *merge*1, pornind din starea inițială și pentru cazul cel mai nefavorabil, este în ordinul lui n^2. Demonstrați aceeași proprietate pentru *find*2 și *merge*2.

Soluție: *find*1 necesită un timp constant și cel mai nefavorabil caz îl reprezintă secvența:

*merge*1(N, N–1); *find*1(N)
*merge*1(N–1, N–2); *find*1(N)
…
*merge*1(N–n+1, N–n); *find*1(N)

În această secvență, *merge*1(N–i+1, N–i) necesită un timp în ordinul lui i. Timpul total este în ordinul lui $1+2+\ldots+n = n(n+1)/2$, deci în ordinul lui n^2. Simetric, *merge*2 necesită un timp constant și cel mai nefavorabil caz îl reprezintă secvența:

*merge*2(N, N–1); *find*2(N)
*merge*2(N–1, N–2); *find*2(N),
…
*merge*2(N–n+1, N–n); *find*2(N)

în care *find*2(i) necesită un timp în ordinul lui i etc.

3.8 De ce am presupus în procedura *merge*3 că $a \neq b$?

3.9 Demonstrați prin inducție că, folosind regula de ponderare (procedura *merge*3), un arbore cu k vârfuri va avea după un număr arbitrar de fuzionări și pornind de la starea inițială, înălțimea maximă $\lfloor \lg k \rfloor$.

Soluție: Proprietatea este adevărată pentru $k = 1$. Presupunem că proprietatea este adevărată pentru $i \leq k-1$ și demonstrăm că este adevărată și pentru k.

Fie T arborele (cu k vârfuri și de înălțime h) rezultat din aplicarea procedurii *merge*3 asupra arborilor T_1 (cu m vârfuri și de înălțime h_1) și T_2 (cu $k-m$ vârfuri și de înălțime h_2). Se observă că cel puțin unul din arborii T_1 și T_2 are cel mult $k/2$ vârfuri, deoarece este imposibil să avem $m > k/2$ și $k-m > k/2$. Presupunând că T_1 are cel mult $k/2$ vârfuri, avem două posibilități:

i) $h_1 \neq h_2 \Rightarrow h \leq \lfloor \lg (k-m) \rfloor \leq \lfloor \lg k \rfloor$

ii) $h_1 = h_2 \Rightarrow h = h_1+1 \leq \lfloor \lg m \rfloor+1 \leq \lfloor \lg (k/2) \rfloor+1 = \lfloor \lg k \rfloor$

3.10 Demonstrați că o serie de n operații *find*2 și *merge*3 necesită, pentru cazul cel mai nefavorabil și pornind de la starea inițială, un timp în ordinul lui $n \log n$.

Indicație: Țineți cont de Exercițiul 3.9 și arătați că timpul este în ordinul lui $n \lg n$. Arătați apoi că baza logaritmului poate fi oarecare, ordinul timpului fiind $n \log n$.

3.11 În locul regulii de ponderare, putem adopta următoarea tactică de fuzionare: rădăcina arborelui cu mai puține vârfuri devine fiu al rădăcinii celuilalt arbore. Comprimarea drumului nu modifică numărul de vârfuri într-un arbore, astfel încât este ușor să memorăm această valoare în mod exact (în cazul folosirii regulii de ponderare, după comprimarea drumului, nu se păstrează înălțimea exactă a unui arbore).

Scrieți o procedură *merge*4 care urmează această tactică și demonstrați un rezultat corespunzător Exercițiului 3.9.

3.12 Găsiți un algoritm pentru a determina dacă un graf neorientat este conex. Folosiți o structură de mulțimi disjuncte.

Indicație: Presupunem că graful este reprezentat printr-o listă de muchii. Considerăm inițial că fiecare vârf formează o submulțime (în acest caz, o componentă conexă a grafului). După fiecare citire a unei muchii $\{a, b\}$ operăm fuzionarea *merge*3(*find*3(a), *find*3(b)), obținând astfel o nouă componentă conexă. Procedeul se repetă, până când terminăm de citit toate muchiile grafului. Graful este conex, dacă și numai dacă tabloul *set* devine constant. Analizați eficiența algoritmului.

În general, prin acest algoritm obținem o partiționare a vârfurilor grafului în submulțimi două câte două disjuncte, fiecare submulțime conținând exact vârfurile câte unei componente conexe a grafului.

3.13 Într-o structură de mulțimi disjuncte, un element x este *canonic*, dacă nu are tată. În procedurile *find*3 și *merge*3 observăm următoarele:

i) Dacă x este un element canonic, atunci informația din *set*[x] este folosită doar pentru a preciza că x este canonic.

ii) Dacă elementul x nu este canonic, atunci informația din $H[x]$ nu este folosită.

Ținând cont de *i)* și *ii)*, modificați procedurile *find*3 și *merge*3 astfel încât, în locul tablourilor *set* și H, să folosiți un singur tablou de N elemente.

Indicație: Utilizați în noul tablou și valori negative.

4. Tipuri abstracte de date

În acest capitol, vom implementa câteva din structurile de date prezentate în Capitolul 3. Utilitatea acestor implementări este dublă. În primul rând, le vom folosi pentru a exemplifica programarea orientată pe obiect prin elaborarea unor noi tipuri abstracte. În al doilea rând, ne vor fi utile ca suport puternic și foarte flexibil pentru implementarea algoritmilor studiați în Capitolele 6-9. Utilizând tipuri abstracte pentru principalele structuri de date, ne vom putea concentra exclusiv asupra algoritmilor pe care dorim să îi programăm, fără a mai fi necesar să ne preocupăm de implementarea structurilor necesare.

Elaborarea fiecărei clase cuprinde două etape, nu neapărat distincte. În prima, vom stabili facilitățile clasei, adică funcțiile și operatorii prin care se realizează principalele operații asociate tipului abstract. De asemenea, vom stabili structura internă a clasei, adică datele membre și funcțiile nepublice. Etapa a doua cuprinde programarea, testarea și depanarea clasei, astfel încât, în final, să avem garanția bunei sale funcționări. Întregul proces de elaborare cuprinde numeroase reveniri asupra unor aspecte deja stabilite, iar fiecare modificare atrage după sine o întreagă serie de alte modificări. Nu vom prezenta toate aceste iterații, deși ele au fost destul de numeroase, ci doar rezultatele finale, comentând pe larg, atât facilitățile clasei, cât și detaliile de implementare. Vom explica astfel și câteva aspecte ale programării orientate pe obiect în limbajul C++, cum sunt clasele parametrice și moștenirea (derivarea). Dorim ca prin această manieră de prezentare să oferim posibilitatea de a înțelege modul de funcționare și utilizare al claselor descrise, chiar dacă anumite aspecte, legate în special de implementare, nu sunt suficient aprofundate.

4.1 Tablouri

În mod surprinzător, începem cu tabloul, structură fundamentală, predefinită în majoritatea limbajelor de programare. Necesitatea de a elabora o nouă structură de acest tip provine din următoarele inconveniente ale tablourilor predefinite, inconveniente care nu sunt proprii numai limbajelor C și C++:

- Numărul elementelor unui tablou trebuie să fie o expresie constantă, fixată în momentul compilării.
- Pe parcursul execuției programului este imposibil ca un tablou să fie mărit sau micșorat după necesități.

- Nu se verifică încadrarea în limitele admisibile a indicilor elementelor tablourilor.
- Tabloul şi numărul elementelor lui sunt două entități distincte. Orice operație cu tablouri (atribuiri, transmiteri de parametri etc) impune specificarea explicită a numărului de elemente ale fiecărui tablou.

4.1.1 Alocarea dinamică a memoriei

Diferența fundamentală dintre tipul abstract pe care îl vom elabora şi tipul tablou predefinit constă în *alocarea dinamică*, în timpul execuției programului, a spațiului de memorie necesar stocării elementelor sale. În limbajul C, alocarea dinamică se realizează prin diversele variante ale funcției `malloc()`, iar eliberarea zonelor alocate se face prin funcția `mfree()`. Limbajul C++ a introdus alocarea dinamică în structura limbajului. Astfel, pentru alocare avem operatorul `new`. Acest operator returnează adresa* zonei de memorie alocată, sau valoarea `0` – dacă alocarea nu s-a putut face. Pentru eliberarea memoriei alocate prin intermediul operatorului `new`, se foloseşte un alt operator numit `delete`. Programul următor exemplifică detaliat funcționarea acestor doi operatori.

```
#include <iostream.h>
#include "intErval.h"

int main( ) {
  // Operatorul new are ca argumente numele unui tip T
  // (predefinit sau definit de utilizator) si dimensiunea
  // zonei care va fi alocata. Valoarea returnata este de
  // tip "pointer la T". Operatorul new returneaza 0 in
  // cazul in care alocarea nu a fost posibila.

  // se aloca o zona de 2048 de intregi
  int *pi = new int [ 2048 ];

  // se aloca o zona de 64 de elemente de tip
  // intErval cu domeniul implicit
  intErval *pi_m = new intErval [ 64 ];

  // se aloca o zona de 8192 de elemente de tip float
  float *pf = new float [ 8192 ];
```

* În limbajul C++, tipul de dată care conține adrese este numit *pointer*. În continuare, vom folosi termenul “pointer”, doar atunci când ne referim la tipul de dată. Termenul “adresă” va fi folosit pentru a ne referi la valoarea datelor de tip pointer.

```
    // De asemenea, operatorul new poate fi folosit pentru
    // alocarea unui singur element de un anumit tip T,
    // precizand eventual si argumentele constructorului
    // tipului respectiv.

    // se aloca un intreg initializat cu 8
    int *i = new int( 8 );

    // se aloca un element de tip intErval
    // cu domeniul admisibil -16, ..., 15
    intErval *m = new intErval( 16, -16 );

    // se aloca un numar real (float) initializat cu 32
    float *f = new float( 32 );


    // Zonele alocate pot fi eliberate oricand si in orice
    // ordine, dar numai prin intermediul pointerului
    // returnat de operatorul new.

    delete [ ] pf;
    delete [ ] pi;
    delete i;
    delete f;
    delete [ ] pi_m;
    delete m;

    return 0;
}
```

Operatorul `new` inițializează memoria alocată prin intermediul constructorilor tipului respectiv. În cazul alocării unui singur element, se invocă constructorul corespunzător argumentelor specificate, iar în cazul alocării unui tablou de elemente, operatorul `new` invocă constructorul implicit pentru fiecare din elementele alocate. Operatorul `delete`, înainte de eliberarea spațiului alocat, va invoca destructorul tipului respectiv. Dacă zona alocată conține un tablou de elemente și se dorește invocarea destructorului pentru fiecare element în parte, operatorul `delete` va fi invocat astfel:

```
delete [ ] pointer;
```

De exemplu, rulând programul

```
#include <iostream.h>

class X {
public:
  X( )  { cout << '*'; }
  ~X( ) { cout << '~'; }
private:
  int x;
};

int main( ) {
  cout << '\n';

  X *p =new X [ 4 ];
  delete p;

  p = new X [ 2 ];
  delete [ ] p;

  cout << '\n';
  return 0;
}
```

constatăm că, în alocarea zonei pentru cele patru elemente de tip `X`, constructorul `X()` a fost invocat de patru ori, iar apoi, la eliberare, destructorul `~X()` doar o singură dată. În cazul zonei de două elemente, atât constructorul cât și destructorul au fost invocați de câte două ori. Pentru unele variante mai vechi de compilatoare C++, este necesar să se specifice explicit numărul elementelor din zona ce urmează a fi eliberată.

În alocarea dinamică, cea mai uzuală eroare este generată de imposibilitatea alocării memoriei. Pe lângă soluția banală, dar extrem de incomodă, de testare a valorii adresei returnate de operatorul `new`, limbajul C++ oferă și posibilitatea invocării, în caz de eroare, a unei funcții definite de utilizator. Rolul acesteia este de a obține memorie, fie de la sistemul de operare, fie prin eliberarea unor zone deja ocupate. Mai exact, atunci când operatorul `new` nu poate aloca spațiul solicitat, el invocă funcția a cărei adresă este dată de variabila globală `_new_handler` și apoi încearcă din nou să aloce memorie. Variabila `_new_handler` este de tip "pointer la funcție de tip `void` fără nici un argument", `void (*_new_handler)()`, valoarea ei implicită fiind `0`.

Valoarea `0` a pointerului `_new_handler` marchează lipsa funcției de tratare a erorii și, în această situație, operatorul `new` va returna `0` ori de câte ori nu poate aloca memoria necesară. Programatorul poate modifica valoarea acestui pointer, fie direct:

```
_new_handler = no_mem;
```

unde `no_mem` este o funcție de tip `void` fără nici un argument,

```
void no_mem( ) {
  cerr << "\n\n no mem. \n\n";
  exit( 1 );
}
```

fie prin intermediul funcției de bibliotecă `set_new_handler`:

```
set_new_handler( no_mem );
```

Toate declarațiile necesare pentru utilizarea pointerului `_new_handler` se găsesc în fișierul header `new.h`.

4.1.2 Clasa `tablou`

Noul tip, numit `tablou`, va avea ca date membre numărul de elemente și adresa zonei de memorie în care sunt memorate acestea. Datele membre fiind `private`, adică inaccesibile din exteriorul clasei, oferim posibilitatea obținerii numărului elementelor tabloului prin intermediul unei funcții membre publice numită `size()`. Iată definiția completă a clasei `tablou`.

```
class tablou {
public:
  // constructorii si destructorul
  tablou( int = 0 );       // constructor (numarul de elemente)
  tablou( const tablou& ); // constructor de copiere
 ~tablou( )  { delete a; } // elibereaza memoria alocata

  // operatori de atribuire si indexare
  tablou& operator =( const tablou& );
  int& operator []( int );

  // returneaza numarul elementelor
  size( ) { return d; }

private:
  int  d; // numarul elementelor (dimensiunea) tabloului
  int *a; // adresa zonei alocate

  // functie auxiliara de initializare
  void init( const tablou& );
};
```

Definițiile funcțiilor membre sunt date în continuare.

```
tablou::tablou( int dim  ) {
  a = 0; d = 0;               // valori implicite
  if ( dim > 0 )              // verificarea dimensiunii
    a = new int [ d = dim ];  // alocarea memoriei
}

tablou::tablou( const tablou& t ) {
  // initializarea obiectului invocator cu t
  init( t );
}

tablou& tablou::operator =( const tablou& t ) {
  if ( this != &t ) { // este o atribuire inefectiva x = x?
    delete a;         // eliberarea memoriei alocate
    init( t );        // initializarea cu t
  }
  return *this;       // se returneaza obiectul invocator
}

void tablou::init( const tablou& t ) {
  a = 0; d = 0;              // valori implicite
  if ( t.d > 0 ) {           // verificarea dimensiunii
    a = new int [ d = t.d ]; // alocarea si copierea elem.
    memcpy( a, t.a, d * sizeof( int ) );
  }
}

int& tablou::operator []( int i ) {
  static int z; // "elementul" tablourilor de dimensiune zero
  return d? a[ i ]: z;
}
```

Fără îndoială că cea mai spectaculoasă definiție este cea a operatorului de indexare `[]`. Acesta permite atât citirea unui element dintr-un `tablou`:

```
tablou x( n );
// ...
cout << x[ i ];
```

cât și modificarea valorii (scrierea) lui:

```
cin >> x[ i ];
```

Facilitățile deosebite ale operatorului de indexare `[]` se datorează tipului valorii returnate. Acest operator nu returnează elementul `i`, ci o referință la elementul `i`, referință care permite accesul atât în scriere, cât și în citire a variabilei de la adresa respectivă.

Clasa `tablou` permite utilizarea tablourilor în care nu există nici un element. Operatorul de indexare `[]` este cel mai afectat de această posibilitate, deoarece

într-un tablou cu zero elemente va fi greu de găsit un element a cărui referință să fie returnată. O soluție posibilă constă în returnarea unui element fictiv, unic pentru toate obiectele de tip `tablou`. În cazul nostru, acest element este variabila locală `static int z`, variabilă alocată static, adică pe toată durata rulării programului.

O atenție deosebită merită și operatorul de atribuire `=`. După cum am precizat în Secțiunea 2.3, structurile pot fi atribuite între ele, membru cu membru. Pentru clasa `tablou`, acest mod de funcționare a operatorului implicit de atribuire este inacceptabil, deoarece generează referiri multiple la aceeași zonă de memorie. Iată un exemplu simplu de ceea ce înseamnă referiri multiple la aceeași zonă de memorie.

Fie `x` și `y` două obiecte de tip tablou. În urma atribuirii `x = y` prin operatorul predefinit `=`, ambele obiecte folosesc aceeași zonă de memorie pentru memorarea elementelor. Dacă unul dintre ele încetează să mai existe, atunci destructorul său îi va elibera zona alocată. În consecință, celălalt va lucra într-o zonă de memorie considerată liberă, zonă care poate fi alocată oricând altui obiect. Prin definirea unui nou operator de atribuire specific clasei `tablou`, obiectele din această clasă sunt atribuite corect, fiecare având propria zonă de memorie în care sunt memorate elementele.

O altă observație relativă la operatorul de atribuire se referă la valoarea returnată. Tipurile predefinite permit concatenarea operatorului de atribuire în expresii de forma

```
i = j = k;
// unde i, j si k sunt variabile de orice tip predefinit
```

Să vedem ce trebuie să facem ca, prin noul operator de atribuire definit, să putem scrie

```
iT = jT = kT;
// iT, jT si kT sunt obiecte de tip tablou
```

Operatorul de atribuire predefinit are asociativitate de dreapta (se evaluează de la dreapta la stânga) și această caracteristică rămâne neschimbată la supraîncărcare. Altfel spus, `iT = jT = kT` înseamnă de fapt `iT = (jT = kT)`, sau `operator =( iT, operator =( jT, kT) )`. Rezultă că operatorul de atribuire trebuie să returneze operandul stâng, sau o referință la acesta. În cazul nostru, operandul stâng este chiar obiectul invocator. Cum în fiecare funcție membră este implicit definit un pointer la obiectul invocator, pointer numit `this` (acesta), operatorul de atribuire va returna o referință la obiectul invocator prin instrucțiunea

```
return *this;
```

Astfel, sintaxa de concatenare poate fi folosită fără nici o restricție.

În definiția clasei tablou a apărut un nou constructor, *constructorul de copiere*

```
tablou( const tablou& )
```

Este un constructor a cărui implementare seamănă foarte mult cu cea a operatorului de atribuire. Rolul său este de a inițializa obiecte de tip `tablou` cu obiecte de același tip. O astfel de operație, ilustrată în exemplul de mai jos, este în mare măsură similară unei copieri.

```
tablou x;
// ...
tablou y = x;   // se invoca constructorul de copiere
```

În lipsa constructorului de copiere, inițializarea se face implicit, adică membru cu membru. Consecințele negative care decurg de aici au fost discutate mai sus.

4.1.3 Clasa parametrică `tablou<T>`

Utilitatea clasei `tablou` este strict limitată la tablourile de întregi, deși un tablou de `float`, `char`, sau de orice alt tip `T`, se manipulează la fel, funcțiile și datele membre fiind practic identice. Pentru astfel de situații, limbajul C++ oferă posibilitatea generării automate de clase și funcții pe baza unor *șabloane* (*template*). Aceste șabloane, numite și *clase parametrice*, respectiv *funcții parametrice*, depind de unul sau mai mulți parametri care, de cele mai multe ori, sunt tipuri predefinite sau definite de utilizator.

Șablonul este o declarație prin care se specifică forma generală a unei clase sau funcții. Iată un exemplul simplu: o funcție care returnează maximul a două valori de tip `T`.

```
template <class T>
T max( T a, T b ) {
  return a > b? a: b;
}
```

Acest șablon se citește astfel: `max()` este o funcție cu două argumente de tip `T`, care returnează maximul celor două argumente, adică o valoare de tip `T`. Tipul `T` poate fi orice tip predefinit, sau definit de utilizator, cu conditia să aibă definit operatorul de comparare `>`, fără de care funcția `max()` nu poate funcționa.

Compilatorul nu generează nici un fel de cod pentru șabloane, până în momentul în care sunt efectiv folosite. De aceea, șabloanele se specifică în fișiere header, fișiere incluse în fiecare program sursă C++ în care se utilizează clasele sau funcțiile parametrice respective*. De exemplu, în funcția

```
void f( int ia, int ib, float fa ) {
  int   m1 = max( ia, ib );
  float m2 = max( ia, fa );
}
```

se invocă funcțiile `int max(int, int)` și `float max(float, float)`, funcții generate automat, pe baza șablonului de mai sus

Conform specificațiilor din Ellis și Stroustrup, *"The Annotated C++ Reference Manual"*, generarea șabloanelor este un proces care nu implică nici un fel de conversii. În consecință, linia

```
  float m2 = max( ia, fa );
```

este eronată. Unele compilatoare nu semnalează această erorare, deoarece invocă totuși conversia lui `ia` din `int` în `float`. Atunci când compilatorul semnalează eroarea, putem declara explicit funcția (vezi și Secțiunea 10.2.3)

```
float max( float, float );
```

declarație care nu mai necesită referirea la șablonul funcției `max()`. Această declarație este, în general, suficientă pentru a genera funcția respectivă pe baza șablonului.

Până când limbajul C++ va deveni suficient de matur pentru a fi standardizat, "artificiile" de programare de mai sus sunt deseori indispensabile pentru utilizarea șabloanelor.

Pentru șabloanele de clase, lucrurile decurg aproximativ în același mod, adică generarea unei anumite clase este declanșată de definițiile întâlnite în program. Pentru clasa parametrică `tablou<T>` definițiile

---

* În prezent sunt utilizate două modele generale pentru instanțierea (generarea) șabloanelor, fiecare cu anumite avantaje și dezavantaje. Reprezentative pentru aceste modele sunt compilatoarele Borland C++ și translatoarele Cfront de la AT&T. Ambele modele sunt compatibile cu plasarea șabloanelor în fișiere header.

```
tablou<float> y( 16 );
tablou<int> x( 32 );
tablou<unsigned char> z( 64 );
```

provoacă generarea clasei `tablou<T>` pentru tipurile `float`, `int` și `unsigned char`. Fișierul header (`tablou.h`) al acestei clase este:

```
#ifndef __TABLOU_H
#define __TABLOU_H

#include <iostream.h>

template <class T>
class tablou {
public:
  // constructorii si destructorul
  tablou( int = 0 );       // constructor (numarul de elemente)
  tablou( const tablou& ); // constructor de copiere
 ~tablou( )  { delete [ ] a; } // elibereaza memoria alocata

  // operatori de atribuire si indexare
  tablou& operator =( const tablou& );
  T& operator []( int );

  // returneaza numarul elementelor
  size( ) { return d; }

  // activarea/dezactivarea verificarii indicilor
  void vOn ( ) { v = 1; }
  void vOff( ) { v = 0; }

protected:
  int  d; // numarul elementelor (dimensiunea) tabloului
  T *a;   // adresa zonei alocate
  char v; // indicator verificare indice

  // functie auxiliara de initializare
  void init( const tablou& );
};

template<class T>
tablou<T>::tablou( int dim ) {
  a = 0; v = 0; d = 0;      // valori implicite
  if ( dim > 0 )            // verificarea dimensiunii
    a = new T [ d = dim ];  // alocarea memoriei
}
```

```
template <class T>
tablou<T>::tablou( const tablou<T>& t ) {
  // initializarea obiectului invocator cu t
  init( t );
}

template <class T>
tablou<T>& tablou<T>::operator =( const tablou<T>& t ) {
  if ( this != &t ) { // este o atribuire inefectiva x = x?
    delete [ ] a;     // eliberarea memoriei alocate
    init( t );        // initializarea cu t
  }
  return *this;       // se returneaza obiectul invocator
}

template<class T>
void tablou<T>::init( const tablou<T>& t ) {
  a = 0; v = 0; d = 0;     // valori implicite
  if ( t.d > 0 ) {         // verificarea dimensiunii
    a = new T [ d = t.d ]; // alocarea si copierea elem.
    for ( int i = 0; i < d; i++ ) a[ i ] = t.a[ i ];
    v = t.v;               // duplicarea indicatorului
  }                        // pentru verificarea indicilor
}

template< class T >
T& tablou<T>::operator []( int i ) {
  static T z;   // elementul returnat in caz de eroare

  if ( d == 0 ) // tablou de dimensiune zero
    return z;

  if ( v == 0 || ( 0 <= i && i < d ) )
    // verificarea indicilor este dezactivata,
    // sau este activata si indicele este corect
    return a[ i ];

  cerr << "\n\ntablou -- " << i
       << ": indice exterior domeniului [0, "
       << ( d - 1 ) << "].\n\n";

  return z;
}
```

Într-o primă aproximare, diferențele față de clasa neparametrică `tablou` sunt următoarele:

- Nivelul de incapsulare `protected` a înlocuit nivelul `private`. Este o modificare necesară procesului de derivare al claselor, prezentat în secțiunile următoare.
- Eliberarea zonei alocate dinamic trebuie să se realizeze prin invocarea destructorului tipului `T` pentru fiecare element. Deci, în loc de `delete a`, este

obligatoriu să scriem `delete [] a` atât în destructor, cât și în operatorul de atribuire. De asemenea, copierea elementelor în funcția `init()` nu se mai poate face global, prin `memcpy()`, ci element cu element, pentru a invoca astfel opratorul de atribuire al tipului `T`.

- Prezența definițiilor funcțiilor membre în fișierul header nu este o greșeală. De fapt, este vorba de șabloanele funcțiilor membre.

Printre inconvenientele tablourilor predefinite am enumerat și imposibilitatea detectării indicilor eronați. După cum se observă, am completat clasa parametrică `tablou<T>` cu funcțiile publice `vOn()` și `vOff()`, prin care se activează, respectiv se dezactivează, verificarea indicilor. În funcție de valoarea logică a variabilei private `v`, valoare stabilită prin funcțiile `vOn()` și `vOff()`, operatorul de indexare va verifica, sau nu va verifica, corectitudinea indicelui. Operatorul de indexare a fost modificat corespunzător.

Pentru citirea și scrierea obiectelor de tip `tablou<T>`, supraîncărcăm operatorii respectivi (`>>` și `<<`) ca funcții nemembre. Convenim ca, în operațiile de citire/scriere, să reprezentăm tablourile în formatul

```
[dimensiune] element1 element2 ...
```

Cei doi operatori pot fi implementați astfel:

```
template <class T>
istream& operator >>( istream& is, tablou<T>& t ) {
  char c;

  // citirea dimensiunii tabloului incadrata de '[' si ']'
  is >> c;
  if ( c != '[' ) { is.clear( ios::failbit ); return is; }
  int n; is >> n; is >> c;
  if ( c != ']' ) { is.clear( ios::failbit ); return is; }

  // modificarea dimensiunii tabloului,
  // evitand copierea elementelor existente
  t.newsize( 0 ).newsize( n );

  // citirea elementelor
  for ( int i = 0; i < n; is >> t[ i++ ] );

  return is;
}
```

```
template <class T>
ostream& operator <<( ostream& os, tablou<T>& t ) {
  int n = t.size( );

  os << " [" << n << "]: ";
  for ( int i = 0; i < n; os << t[ i++ ] << ' ' );

  return os;
}
```

Acești operatori sunt utilizabili doar dacă obiectelor de tip `T` li se pot aplica operatorii de extragere/inserare `>>`, respectiv `<<`. În caz contrar, orice încercare de a aplica obiectelor de tip `tablou<T>` operatorii mai sus definiți, va fi semnalata ca eroare la compilarea programului.

Operatorul de extragere (citire) `>>` prezintă o anumită particularitate față de celelalte funcții care operează asupra tablourilor: trebuie să modifice chiar dimensiunea tabloului. Două variante de a realiza această operație, dintre care una prin intermediul funcției `newsize( )`, sunt discutate în Exercițiile 4.2 și 4.3.

Marcarea erorilor la citire se realizează prin modificarea corespunzătoare a stării `istream`-ului prin

```
is.clear( ios::failbit );
```

După cum am precizat în Secțiunea 2.3.2, starea unui `istream` se poate testa printr-un simplu `if ( cin >> ... )`. Odată ce un `istream` a ajuns într-o stare de eroare, nu mai răspunde la operatorii respectivi, decât după ce este readus la starea normală de utilizare prin instrucțiunea

```
is.clear();
```

4.2 Stive, cozi, heap-uri

Stivele, cozile și heap-urile sunt, în esență, tablouri manipulate altfel decât prin operatorul de indexare. Acesată afirmație contrazice aparent definițiile date în Capitolul 3. Aici se precizează că stivele și cozile sunt liste liniare în care inserările/extragerile se fac conform unor algoritmi particulari, iar heap-urile sunt arbori binari compleți. Tot în Capitolul 3 am arătat că reprezentarea cea mai comodă pentru toate aceste structuri este cea secvențială, bazată pe tablouri.

În terminologia specifică programării orientate pe obiect, spunem că tipurile `stiva<T>`, `coada<T>` și `heap<T>` sunt *derivate* din tipul `tablou<T>`, sau că *moștenesc* tipul `tablou<T>`. Tipul `tablou<T>` se numește *tip de bază* pentru

tipurile `stiva<T>`, `coada<T>` și `heap<T>`. Prin moștenire, limbajul C++ permite atât crearea unor subtipuri ale tipului de bază, cât și crearea unor tipuri noi, diferite de tipul de bază. Stivele, cozile și heap-urile vor fi tipuri noi, diferite de tipul de bază tablou. Posibilitatea de a crea subtipuri prin derivare, o facilitate deosebit de puternică a programării orientate pe obiect și a limbajului C++, va fi exemplificată în Secțiunile 11.1 și 10.2.

4.2.1 Clasele `stiva<T>` și `coada<T>`

Clasa `stiva<T>` este un tip nou, derivat din clasa `tablou<T>`. În limbajul C++, derivarea se indică prin specificarea claselor de bază (pot fi mai multe!), imediat după numele clasei.

```
template <class T>
class stiva: private tablou<T> {
// ....
};
```

Fiecare clasă de bază este precedată de atributul `public` sau `private`, prin care se specifică modalitatea de moștenire. O clasă derivată `public` este un subtip al clasei de bază, iar una derivată `private` este un tip nou, distinct față de tipul de bază.

Clasa derivată moștenește toți membrii clasei de bază, cu excepția constructorilor și destructorilor, dar nu are acces la membrii `private` ai clasei de bază. Atunci când este necesar, acest incovenient poate fi evitat prin utilizarea în clasa de bază a nivelului de acces `protected` în locul celui `private`. Membrii `protected` sunt membri privați, dar accesibili claselor derivate. Nivelul de acces al membrilor moșteniți se modifică prin derivare astfel:

- Membrii neprivați dintr-o clasă de bază publică își păstrează nivelele de acces și în clasa derivată.
- Membrii neprivați dintr-o clasă de bază privată devin membri `private` în clasa derivată.

Revenind la clasa `stiva<T>`, putem spune că moștenește de la clasa de bază `tablou<T>` membrii

```
int d;
T  *a;
```

ca membri `private`, precum și cei doi operatori (publici în clasa `tablou<T>`)

```
tablou& operator =( const tablou& );
T& operator []( int );
```

tot ca membri `private`.

Pe baza celor de mai sus, se justifică foarte simplu faptul că prin derivarea privată se obțin tipuri noi, total distincte față de tipul de bază. Astfel, nu este disponibilă nici una din facilitățile clasei de bază `tablou<T>` în exteriorul clasei `stiva<T>`, existența clasei de bază fiind total ascunsă utilizatorului. În schimb, pentru implementarea propriilor facilități, clasa `stiva<T>` poate folosi din plin toți membrii clasei `tablou<T>`. Prin derivarea `private`, realizăm deci o *reutilizare* a clasei de bază.

Definirea unei stive derivată din tablou se realizează astfel (fișierul `stiva.h`):

```
#ifndef __STIVA_H
#define __STIVA_H

#include <iostream.h>
#include "tablou.h"

template <class T>
class stiva: private tablou<T> {
public:
  stiva( int d ):  tablou<T>( d ) { s = -1; }

  push( const T& );
  pop (       T& );

private:
  int s;  // indicele ultimului element inserat
};

template <class T>
stiva<T>::push( const T& v ) {
  if ( s >= d - 1 ) return 0;
  a[ ++s ] = v;     return 1;
}

template <class T>
stiva<T>::pop( T& v ) {
  if ( s < 0 )  return 0;
  v = a[ s-- ]; return 1;
}

#endif
```

Înainte de a discuta detaliile de implementare, să remarcăm o anumită inconsecvență apărută în definiția funcției `pop()` din Secțiunea 3.1.1. Această funcție returnează fie elementul din vârful stivei, fie un mesaj de eroare (atunci când stiva este vidă). Desigur că nu este un detaliu deranjant atât timp cât ne

interesează doar algoritmul. Dar, cum implementăm efectiv această funcție, astfel încât să cuprindem ambele situații? Întrebarea poate fi formulată în contextul mult mai general al tratării excepțiilor. Rezolvarea unor cazuri particulare, a excepțiilor de la anumite reguli, problemă care nu este strict de domeniul programării, poate da mai puține dureri de cap prin aplicarea unor principii foarte simple. Iată, de exemplu, un astfel de principiu formulat de Winston Churchill: "Nu mă intrerupeți în timp ce întrerup".

Tratarea excepțiilor devine o chestiune foarte complicată, mai ales în cazul utilizării unor funcții sau obiecte dintr-o bibliotecă. Autorul unei biblioteci de funcții (obiecte) poate detecta excepțiile din timpul execuției dar, în general, nu are nici o idee cum să le trateze. Pe de altă parte, utilizatorul bibliotecii știe ce să facă în cazul apariției unor excepții, dar nu le poate detecta. Noțiunea de excepție, noțiune acceptată de Comitetul de standardizare ANSI C++, introduce un mecanism consistent de rezolvare a unor astfel de situații. Ideea este ca, în momentul când o funcție detectează o situație pe care nu o poate rezolva, să semnaleze (`throw`) o excepție, cu speranța că una din funcțiile (direct sau indirect) invocatoare va rezolva apoi problema. O funcție care este pregătită pentru acest tip de evenimente își va anunța în prealabil disponibilitatea de a trata (`catch`) excepții.

Mecanismul schițat mai sus este o alternativă la tehnicile tradiționale, atunci când acestea se dovedesc a fi inadecvate. El oferă o cale de separare explicită a secvențelor pentru tratarea erorilor de codul propriu-zis, programul devenind astfel mai clar și mult mai ușor de întreținut. Din păcate, la nivelul anului 1994, foarte puține compilatoare C++ implementează complet mecanismul `throw`–`catch`. Revenim de aceea la "stilul clasic", stil independent de limbajul de programare folosit. Uzual, la întâlnirea unor erori se acționează în unul din următoarele moduri:

- Se termină programul.
- Se returnează o valoare reprezentând "eroare".
- Se returnează o valoare legală, programul fiind lăsat într-o stare ilegală.
- Se invocă o funcție special construită de programator pentru a fi apelată în caz de eroare.

Terminarea programului se realizează prin revenirea din funcția `main()`, sau prin invocarea unei funcții de bibliotecă numită `exit()`. Valoarea returnată de `main()`, precum și argumentul întreg al funcției `exit()`, este interpretat de sistemul de operare ca un *cod de retur* al programului. Un cod de retur nul (zero) semnifică executarea corectă a programului.

Până în prezent, am utilizat tratarea excepțiilor prin terminarea programului în clasa `intErval`. Un alt exemplu de tratare a excepțiilor se poate remarca la operatorul de indexare din clasa `tablou<T>`. Aici am utilizat penultima alternativă

din cele patru enunțate mai sus: valoarea returnată este legală, dar programul nu a avut posibilitatea de a trata eroarea.

Pentru stivă și, de fapt, pentru multe din structurile implementate aici și susceptibile la situații de excepție, am ales varianta a doua: returnarea unei valori reprezentând "eroare". Pentru a putea distinge cât mai simplu situațiile normale de cazurile de excepție, am convenit ca funcția `pop()` să transmită elementul din vârful stivei prin intermediul unui argument de tip referință, valoarea returnată efectiv de funcție indicând existența sau inexistența acestui element. Astfel, secvența

```
while( s.pop( v ) ) {
  // ...
}
```

se execută atât timp cât în stiva `s` mai sunt elemente, variabila `v` având de fiecare dată valoarea elementului din vârful stivei. Funcția `push()` are un comportament asemănător, secvența

```
while( s.push( v ) ) {
  // ...
}
```

executându-se atâta timp cât în stivă se mai pot insera elemente.

În continuare, ne propunem să analizăm mai amănunțit contribuția clasei de bază `tablou<T>` în funcționarea clasei `stiva<T>`. Să remarcăm mai întâi invocarea constructorului tipului de bază pentru inițializarea datelor membre moștenite, invocare realizată prin *lista de inițializare a membrilor*:

```
stiva( int d ): tablou<T>( d ) { s = -1; }
```

Utilizarea acestei sintaxe speciale se datorează faptului că execuția oricărui constructor se face în două etape. Într-o primă etapă, *etapă de inițializare*, se invocă constructorii datelor membre moștenite de la clasele de bază, conform listei de inițializare a membrilor. În a doua etapă, numită *etapă de atribuire*, se execută corpul propriu-zis al constructorului. Necesitatea unei astfel de etapizări se justifică prin faptul că inițializarea membrilor moșteniți trebuie rezolvată în mod unitar de constructorii proprii, și nu de cel al clasei derivate. Dacă lista de inițializare a membrilor este incompletă, atunci, pentru membrii rămași neinițializați, se invocă constructorii impliciți. De asemenea, tot în etapa de inițializare se vor invoca constructorii datelor membre de tip clasă și se vor inițializa datele membre de tip `const` sau referință.

Continuând analiza contribuției tipului de bază `tablou<T>`, să remarcăm că în clasa `stiva<T>` nu s-au definit constructorul de copiere, operatorul de atribuire și

destructorul. Inițializarea și atribuirea obiectelor de tip stivă cu obiecte de același tip, precum și distrugerea acestora, se realizează totuși corect, datele membre moștenite de la `tablou<T>` fiind manipulate de funcțiile membre ale acestui tip. În funcția

```
void f( ) {
  stiva<int> x( 16 );
  stiva<int> y = x;
  x = y;
}
```

inițializarea lui `y` cu `x` se face membru cu membru, pentru datele proprii clasei `stiva<T>` (întregul `top`), și prin invocarea constructorului de copiere al clasei `tablou<T>`, pentru inițializarea datelor membre moștenite (întregul `d` și adresa `a`). Atribuirea `x = y` se efectuează membru cu membru, pentru datele proprii, iar pentru cele moștenite, prin invocarea operatorului de atribuire al clasei `tablou<T>`. La terminarea funcției, obiectele `x` și `y` vor fi distruse prin invocarea destructorilor în ordinea inversă a invocării constructorilor, adică destructorul clasei `stiva<T>` (care nu a fost precizat pentru că nu are de făcut nimic) și apoi destructorul clasei de bază `tablou<T>`.

Implementarea clasei `coada<T>` se face pe baza precizărilor din Secțiunea 3.1.2, direct prin modificarea definiției clasei `stiva<T>`. În locul indicelui `top`, vom avea două date membre, și anume indicii `head` și `tail`, iar funcțiile membre `push()` și `pop()` vor fi înlocuite cu `ins_q()`, respectiv `del_q()`. Ca exercițiu, vă propunem să realizați implementarea efectivă a acestei clase.

4.2.2 Clasa heap<T>

Vom utiliza structura de heap descrisă în Secțiunea 3.4 pentru implementarea unei clase definită prin operațiile de inserare a unei valori și de extragere a maximului. Clasa parametrică `heap<T>` seamănă foarte mult cu clasele `stiva<T>` și `coada<T>`. Diferențele apar doar la implementarea operațiilor de inserare în heap și de extragere a maximului. Definiția clasei `heap<T>` este:

```
#ifndef __HEAP_H
#define __HEAP_H

#include <iostream.h>
#include <stdlib.h>
#include "tablou.h"
```

```
template <class T>
class heap: private tablou<T> {
public:
  heap( int d ): tablou<T>( d ) { h = -1; }
  heap( const tablou<T>& t ): tablou<T>( t )
    { h = t.size( ) - 1; make_heap( ); }

  insert    ( const T& );
  delete_max(       T& );

protected:
  int h;      // indicele ultimului element din heap

  void percolate( int );
  void sift_down( int );
  void make_heap( );
};

template <class T>
heap<T>::insert( const T& v ) {
  if ( h >= d - 1 ) return 0;
  a[ ++h ] = v; percolate( h );
  return 1;
}

template <class T>
heap<T>::delete_max( T& v ) {
  if ( h < 0 ) return 0;
  v = a[ 0 ];
  a[ 0 ] = a[ h-- ]; sift_down( 0 );
  return 1;
}

template <class T>
void heap<T>::make_heap( ) {
  for ( int i = (h + 1) / 2; i >= 1; sift_down( --i ) );
}

template <class T>
void heap<T>::percolate( int i ) {
  T *A = a - 1;  // a[ 0 ]     este A[ 1 ], ...,
                 // a[ i - 1 ] este A[ i ]
  int k = i + 1, j;

  do {
    j = k;
    if ( j > 1 && A[ k ] > A[ j/2 ] ) k = j/2;
    T tmp = A[ j ]; A[ j ] = A[ k ]; A[ k ] = tmp;
  } while ( j != k );
}
```

```
template <class T>
void heap<T>::sift_down( int i ) {
  T *A = a - 1;  // a[ 0 ]     este A[ 1 ], ...,
                 // a[ n - 1 ] este A[ n ]
  int n = h + 1, k = i + 1, j;

  do {
    j = k;
    if ( 2*j <= n && A[ 2*j   ] > A[ k ] ) k = 2*j;
    if ( 2*j <  n && A[ 2*j+1 ] > A[ k ] ) k = 2*j+1;
    T tmp = A[ j ]; A[ j ] = A[ k ]; A[ k ] = tmp;
  } while ( j != k );
}

#endif
```

Procedurile `insert()` și `delete_max()` au fost adaptate stilului de tratare a excepțiilor prezentat în secțiunea precedentă: ele returnează valorile logice *true* sau *false*, după cum operațiile respective sunt, sau nu sunt posibile.

Clasa `heap<T>` permite crearea unor heap-uri cu elemente de cele mai diverse tipuri: `int`, `float`, `long`, `char` etc. Dar încercarea de a defini un heap pentru un tip nou `T`, definit de utilizator, poate fi respinsă chiar în momentul compilării, dacă acest tip nu are definit operatorul de comparare `>`. Acest operator, a cărui definire rămâne în sarcina proiectantului clasei `T`, trebuie să returneze *true* (o valoare diferită de `0`) dacă argumentele sale sunt în relația `>` și *false* (adică `0`) în caz contrar. Pentru a nu fi necesară și definirea operatorului `<`, în implementarea clasei `heap<T>` am folosit numai operatorul `>`.

Vom exemplifica utilizarea clasei `heap<T>` cu un operator `>` diferit de cel predefinit prin intermediul clasei `intErval`. Deși clasa `intErval` nu are definit operatorul `>`, programul următor “trece” de compilare și se execută (aparent) corect.

```
#include "intErval.h"
#include "heap.h"

// dimensiunea heap-ului, margine superioara in intErval
const SIZE = 128;

int main( ) {
  heap<intErval> hi( SIZE );
  intErval v( SIZE );
```

```
  cout << "Inserare in heap (^Z/#" << (SIZE - 1) << ")\n... ";
  while ( cin >> v ) {
    hi.insert( v );
    cout << "... ";
  }
  cin.clear( );

  cout << "Extragere din heap\n";
    while ( hi.delete_max( v ) ) cout << v << '\n';

  return 0;
}
```

Justificarea corectitudinii sintactice a programului de mai sus constă în existența operatorului de conversie de la `intErval` la `int`. Prin această conversie, compilatorul rezolvă compararea a două valori de tip `intErval` (pentru operatorul `>`), sau a unei valori `intErval` cu valoarea `0` (pentru operatorul `!=`) folosind operatorii predefiniți pentru argumente de tip întreg. Utilizând același operator de conversie de la `intErval` la `int`, putem defini foarte comod un operator `>`, prin care heap-ul să devină un min-heap. Noul operator `>` este practic negarea relației uzuale `>`:

```
// Operatorul > pentru min-heap
int operator >( const intErval&  a, const intErval& b ) {
  return a < b;
}
```

La compilarea programului de mai sus, probabil că ați observat un mesaj relativ la invocarea funcției "non-`const`" `intErval::operator int()` pentru un obiect `const` în funcția `heap<T>::insert()`. Iată despre ce este vorba. Următorul program generează exact același mesaj:

```
#include "intErval.h"

int main( ) {
  intErval       x1;
  const intErval x2( 20, 10 );

  x1 = x2;
  return 0;
}
```

Deși nu este invocat explicit, operatorul de conversie la `int` este aplicat variabilei constante `x2`. Înainte de a discuta motivul acestei invocări, să ne oprim puțin asupra manipulării obiectelor constante. Pentru acest tip de variabile (variabile constante!), așa cum este `x2`, se invocă doar funcțiile membre declarate explicit `const`, funcții care nu modifică obiectul invocator. O astfel de funcție fiind și

operatorul de conversie `intErval::operator int()`, va trebui să-i completăm definiția din clasa `intErval` cu atributul `const`:

```
operator int( ) const { return v; }
```

Același efect îl are și definirea non-`const` a obiectului `x2`, dar scopul nu este de a elimina mesajul, ci de a înțelege (și de a elimina) cauza lui.

Atribuirea `x1 = x2` ar trebui rezolvată de operatorul de atribuire generat automat de compilator, pentru fiecare clasă. În cazul nostru, acest operator nu se invocă, deoarece atribuirea poate fi rezolvată numai prin intermediul funcțiilor membre explicit definite:

- `x2` este convertit la `int` prin `operator int( )`, conversie care generează și mesajul discutat mai sus
- Rezultatul conversiei este atribuit lui `x1` prin `operator =(int)`.

Din păcate, rezultatul atribuirii este incorect. În loc ca `x2` să fie copiat în `x1`, va fi actualizată doar valoarea `v` a lui `x1` cu valoarea `v` lui `x2`. Evident că, în exemplul de mai sus, `x1` va semnala depășirea domeniului său.

Soluția pentru eliminarea acestei aparente anomalii, generate de interferența dintre `operator int( )` și `operator =(int)`, constă în definirea explicită a operatorului de atribuire pentru obiecte de tip `intErval`:

```
intErval& intErval::operator =( const intErval& s ) {
  min = s.min; v = s.v; max = s.max;
  return *this;
}
```

După ce am clarificat particularitățile obiectelor constante, este momentul să adaptăm corespunzător și clasa `tablou<T>`. Orice clasă frecvent utilizată – și `tablou<T>` este una din ele – trebuie să fie proiectată cu grijă, astfel încât să suporte inclusiv lucrul cu obiecte constante. Vom adăuga în acest scop atributul `const` funcției membre `size()`:

```
  size( ) const { return d; }
```

În plus, mai adăugăm și un nou operator de indexare:

```
const T& operator []( int ) const;
```

Particularitatea acestuia constă doar în tipul valorii returnate, `const T&`, valoare imposibil de modificat. Consistența declarației `const`, asociată operatorului de indexare, este dată de către proiectantul clasei și nu poate fi verificată semantic de către compilator. O astfel de declarație poate fi atașată chiar și operatorului de

indexare obișnuit (cel non-`const`), căci el nu modifică nici una din datele membre ale clasei `tablou<T>`. Ar fi însă absurd, deoarece tabloul se modifică de fapt prin modificarea elementelor sale.

4.3 Clasa `lista<E>`

Structurile prezentate până acum sunt de fapt liste implementate secvențial, diferențiate prin particularitățile operațiilor de inserare și extragere. În cele ce urmează, ne vom concentra asupra unei implementări înlănțuite a listelor, prin alocarea dinamică a memoriei.

Ordinea nodurilor unei liste se realizează prin completarea informației propriu-zise din fiecare nod, cu informații privind localizarea nodului următor și eventual a celui precedent. Informațiile de localizare, numite legături sau adrese, pot fi, în funcție de modul de implementare ales (vezi Secțiunea 3.1), indici într-un tablou, sau adrese de memorie. În cele ce urmează, fiecare nod va fi alocat dinamic prin operatorul `new`, legăturile fiind deci adrese.

Informația din fiecare nod poate fi de orice tip, de la un număr întreg sau real la o structură oricât de complexă. De exemplu, pentru reprezentarea unui graf prin lista muchiilor, fiecare nod conține cele două extremități ale muchiei și lungimea (ponderea) ei. Limbajul C++ permite implementarea structurii de nod prin intermediul claselor parametrice astfel:

```
template <class E>
class nod {
// ...
  E       val;  // informatia propriu-zisa
  nod<E> *next; // adresa nodului urmator
};
```

Operațiile elementare, cum sunt parcurgerile, inserările sau ștergerile, pot fi implementate prin intermediul acestei structuri astfel:

- Parcurgerea nodurilor listei:

```
nod<E> *a;      // adresa nodului actual
// ...
while ( a ) {   // adresa ultimului element are valoarea 0
  // ...          prelucrarea informatiei a->val
  a = a->next;  // notatie echivalenta cu a = (*a).next
}
```

- Inserarea unui nou nod în listă:

```
nod<E> *a;      // adresa nodului dupa care se face inserarea
nod<E> *pn;     // adresa nodului de inserat
// ...
pn->next = a->next;
a->next  = pn;
```

- Ștergerea unui nod din listă (operație care necesită cunoașterea nu numai a adresei elementului de eliminat, ci și a celui anterior):

```
nod<E> *a;          // adresa nodului de sters
nod<E> *pp;         // adresa nodului anterior lui a
// ...
pp->next = a->next; // stergerea propriu-zisa

// ...
// eliberarea spatiului de memorie alocat nodului de
// adresa a, nod tocmai eliminat din lista
```

Structura de nod este suficientă pentru manipularea listelor cu elemente de tip E, cu condiția să cunoaștem primul nod:

```
nod<E> head;  // primul nod din lista
```

Există totuși o listă imposibil de tratat prin intermediul acestei implementări, și anume lista vidă. Problema de rezolvat este oarecum paradoxală, deoarece variabila head, primul nod din listă, trebuie să reprezinte un nod care nu există. Se pot găsi diverse soluții particulare, dependente de tipul și natura informațiilor. De exemplu, dacă informațiile sunt valori pozitive, o valoare negativă ar putea reprezenta un nod inexistent. O altă soluție este adăugarea unei noi date membre pentru validarea existenței nodului curent. Dar este inacceptabil ca pentru un singur nod și pentru o singură situație să încărcăm toate celelalte noduri cu încă un câmp.

Imposibilitatea reprezentării listelor vide nu este rezultatul unei proiectări defectuoase a clasei nod<E>, ci al confuziei dintre listă și nodurile ei. Identificând lista cu adresa primului ei nod și adăugând funcțiile uzuale de manipulare (inserări, ștergeri etc), obținem tipul abstract lista<E> cu elemente de tip E:

```
template <class E>
class lista {
// ...
private:
  nod<E> *head;  // adresa primul nod din lista
};
```

Conform principiilor de încapsulare, manipularea obiectelor clasei abstracte lista<E> se face exclusiv prin intermediul funcțiilor membre, structura internă a

listei şi, desigur, a nodurilor, fiind invizibilă din exterior. Contează doar tipul informaţiilor din listă şi nimic altceva. Iată de ce clasa `nod<E>` poate fi în întregime nepublică:

```
template <class E>
class nod {
  friend class lista<E>;
  // ...
protected:
  nod( const E& v ): val( v ) { next = 0; }

  E       val;  // informatia propriu-zisa
  nod<E> *next; // adresa nodului urmator
};
```

În lipsa declaraţiei `friend`, obiectele de tip `nod<E>` nici măcar nu pot fi definite, datorită lipsei unui constructor `public`. Prin declaraţia `friend` se permite accesul clasei `lista<E>` la toţi membrii privaţi ai clasei `nod<E>`. Singurul loc în care putem utiliza obiectele de tip `nod<E>` este deci domeniul clasei `lista<E>`.

Înainte de a trece la definirea funcţiilor de manipulare a listelor, să remarcăm un aspect interesant la constructorul clasei `nod<E>`. Iniţializarea membrului `val` cu argumentul `v` nu a fost realizată printr-o atribuire `val = v`, ci invocând constructorul clasei `E` prin lista de iniţializare a membrilor:

```
nod( const E& v ): val( v ) { // ... }
```

În acest context, atribuirea este ineficientă, deoarece `val` ar fi iniţializat de două ori: o dată în faza de iniţializare prin constructorul implicit al clasei `E`, iar apoi, în faza de atribuire, prin invocarea operatorului de atribuire.

Principalele operaţii asupra listelor sunt inserarea şi parcurgerea elementelor. Pentru a implementa parcurgerea, să ne amintim ce înseamnă parcurgerea unui tablou – pur şi simplu un indice şi un operator de indexare:

```
tablou<int> T( 32 );
T[ 31 ] = 1;
```

În cazul listelor, locul indicelui este luat de elementul curent. Ca şi indicele, care nu este memorat în clasa tablou, acest element curent nu are de ce să facă parte din structura clasei `lista<T>`. Putem avea oricâte elemente curente, corespunzătoare oricâtor parcurgeri, tot aşa cum un tablou poate fi adresat prin oricâţi indici. Analogia tablou-listă se sfârşeşte aici. Locul operatorului de indexare `[]` nu este luat de o funcţie membră, ci de o clasă specială numită `iterator<E>`.

Într-o variantă minimă, datele membre din clasa `iterator<E>` sunt:

```
template <class E>
class iterator {
// ...
private:
  nod<E>* const *phead;
  nod<E>        *a;
};
```

adică adresa nodului actual (curent) și adresa adresei primului element al listei. De ce adresa adresei? Pentru ca iteratorul să rămână funcțional și în situația eliminării primului element din listă. Operatorul `()`, numit în terminologia specifică limbajului C++ *iterator*, este cel care implementează efectiv operația de parcurgere

```
template <class E>
iterator<E>::operator ()( E& v ) {
  if( a ) { v = a->val;  a = a->next; return 1; }
  else    { if( *phead ) a = *phead;  return 0; }
}
```

Se observă că parcurgerea este circulară, adică, odată ce elementul actual a ajuns la sfârșitul listei, el este inițializat din nou cu primul element, cu condiția ca lista să nu fie vidă. Atingerea sfârșitului listei este marcată prin returnarea valorii *false*. În caz contrar, valoarea returnată este *true*, iar elementul curent este "returnat" prin argumentul de tip referință la `E`. Pentru exemplificare, operatorul de inserare în `ostream` poate fi implementat prin clasa `iterator<E>` astfel:

```
template <class E>
ostream& operator <<( ostream& os, const lista<E>& lista ) {
  E v; iterator<E> l = lista;

  os << " { ";
  while ( l( v ) ) os << v << ' ';
  os << "} ";

  return os;
}
```

Inițializarea iteratorului `l`, realizată prin definiția `iterator<E> l = lista`, este implementată de constructorul

```
template <class E>
iterator<E>::iterator( const lista<E>& l ) {
  phead = &l.head;
  a = *phead;
}
```

Declarația `const` a argumentului `lista<E>& l` semnifică faptul că `l`, împreună cu datele membre, este o variabilă read-only (constantă) în acest constructor. În consecință, `*phead` trebuie să fie constant, adică definit ca

```
nod<E>* const *phead;
```

Aceeași inițializare mai poate fi realizată și printr-o instrucțiune de atribuire `l = lista`, operatorul corespunzător fiind asemănător celui de mai sus:

```
template <class E>
iterator<E>& iterator<E>::operator =( const lista<E>& l ) {
  phead = &l.head;
  a = *phead;

  return *this;
}
```

Pentru a putea defini un iterator neinițializat, se va folosi constructorul implicit (fără nici un argument):

```
template <class E>
iterator<E>::iterator( ) {
  phead = 0;
  a = 0;
}
```

În finalul discuției despre clasa `iterator<E>`, vom face o ultimă observație. Această clasă trebuie să aibă acces la membrii privați din clasele `nod<E>` și `lista<E>`, motiv pentru care va fi declarată `friend` în ambele.

În sfârșit, putem trece acum la definirea completă a clasei `lista<E>`. Funcția `insert()` inserează un nod înaintea primului element al listei.

```
template <class E>
lista<E>& lista<E>::insert( const E& v ) {
  nod<E> *pn = new nod<E>( v );
  pn->next = head; head = pn;

  return *this;
}
```

O altă funcție membră, numită `init()`, este invocată de către constructorul de copiere și de către operatorul de atribuire, pentru inițializarea unei liste noi cu o alta, numită listă `sursa`.

```
template <class E>
void lista<E>::init( const lista<E>& sursa ) {
  E v; iterator<E> s = sursa;

  for ( nod<E> *tail = head = 0; s( v ); ) {
    nod<E> *pn = new nod<E>( v );
    if ( !tail ) head = pn; else tail->next = pn;
    tail = pn;
  }
}
```

Funcția `reset()` elimină rând pe rând toate elementele listei:

```
template <class E>
void lista<E>::reset( ) {
  nod<E> *a = head;

  while( a ) {
    nod<E> *pn = a->next;
    delete a;
    a = pn;
  }
  head = 0;
}
```

Instrucțiunea `head = 0` are, aparent, același efect ca întreaga funcție `reset()`, deoarece lista este redusă la lista vidă. Totuși, această instrucțiune nu se poate substitui întregii funcții, deoarece elementele listei ar rămâne alocate, fără să existe posibilitatea de a recupera spațiul alocat.

Declarațiile claselor `nod<E>`, `lista<E>` și `iterator<E>`, în forma lor completă, sunt următoarele:

```
template <class E>
class nod {
  friend class lista<E>;
  friend class iterator<E>;

protected:
  nod( const E& v ): val( v ) { next = 0; }

  E       val;  // informatia propriu-zisa
  nod<E> *next; // adresa nodului urmator
};
```

```
template <class E>
class lista {
  friend class iterator<E>;
public:
  lista(  ) { head = 0; }
  lista( const lista<E>& s ) { init( s ); }
  ~lista( ) { reset( ); }

  lista& operator =( const lista<E>& );
  lista& insert( const E& );

private:
  nod<E> *head;  // adresa primul nod din lista

  void init( const lista<E>& );
  void reset( );
};

template <class E>
class iterator {
public:
  iterator( );
  iterator( const lista<E>& );

  operator ()( E& );
  iterator<E>& operator =( const lista<E>& );

private:
  nod<E>* const *phead;
  nod<E>        *a;
};
```

4.4 Exerciții

4.1 În cazul alocării dinamice, este mai rentabil ca memoria să se aloce în blocuri mici sau în blocuri mari?

Soluție: Rulați următorul program. Atenție, stiva programului trebuie să fie suficient de mare pentru a "rezista" apelurilor recursive ale funcției `alocareDinmica()`.

```
#include <iostream.h>

static int nivel;
static int raport;
```

```
void alocareDinamica( unsigned n ) {
  ++nivel;
  char *ptr = new char[ n ];
  if ( ptr )
    alocareDinamica( n );

  // memoria libera este epuizata
  delete ptr;
  if ( !raport++ )
    cout << "\nMemoria libera a fost epuizata. "
      << "S-au alocat "
      << (long)nivel * n * sizeof( char ) / 1024 << 'K'
      << ".\nNumarul de apeluri " << nivel
      << "; la fiecare apel s-au alocat "
      << n * sizeof( char ) << " octeti.\n";
}


main( ) {
  for ( unsigned i = 1024; i > 32; i /= 2 ) {
    nivel = 1; raport = 0;
    alocareDinamica( 64 * i - 1 );
  }

  return 1;
}
```

Rezultatele obținute sunt clar în favoarea blocurilor mari. Explicația constă în faptul că fiecărui bloc alocat i se adaugă un antet necesar gestionării zonelor ocupate și a celor libere, zone organizate în două liste înlănțuite.

4.2 Explicați rezultatele programului de mai jos.

```
#include <iostream.h>
#include "tablou.h"

int main( ) {
  tablou<int> y( 12 );

  for ( int i = 0, d = y.size( ); i < d; i++ )
    y[ i ] = i;

  cout << "\nTabloul y      : " << y;
  y = 8;
  cout << "\nTabloul y      : " << y;

  cout << '\n';
  return 0;
}
```

Soluție: Elementul surprinzător al acestui program este instrucțiunea de atribuire `y = 8`. Surpinzător, în primul rând, deoarece ea “trece” de compilare, deși nu s-a

definit operatorul de atribuire corespunzător. În al doilea rând, instrucțiunea `y = 8` surprinde prin efectele execuției sale: tabloul `y` are o altă dimensiune și un alt conținut. Explicația este dată de o convenție a limbajului C++, prin care un constructor cu un singur argument este folosit și ca operator de conversie de la tipul argumentului, la tipul clasei respective. În cazul nostru, tabloului `y` i se atribuie un tablou temporar de dimensiune 8, generat prin invocarea constructorului clasei `tablou<T>` cu argumentul 8. S-a realizat astfel modificarea dimensiunii tabloului `y`, dar cu prețul pierderii conținutului inițial.

4.3 Exercițiul de mai sus conține o soluție pentru modificarea dimensiunii obiectelor de tip `tablou<T>`. Problema pe care o punem acum este de a rezolva problema, astfel încât conținutul tabloului să nu se mai piardă.

Soluție: Iată una din posibilele implementări:

```
template< class T >
tablou<T>& tablou<T>::newsize( int dN ) {
  T *aN = 0;              // noua adresa

  if ( dN > 0 ) {
    aN = new T [ dN ];    // alocarea dinamica a memoriei
    for ( int i = d < dN? d: dN; i--; )
      aN[ i ] = a[ i ];   // alocarea dinamica a memoriei
  }
  else
    dN = 0;

  delete [ ] a;           // eliberarea vechiului spatiu
  d = dN; a = aN;         // redimensionarea obiectului

  return *this;
}
```

4.4 Implementați clasa parametrică `coada<T>`.

Soluție: Conform celor menționate la sfârșitul Secțiunii 4.2.1, ne vom inspira de la structura clasei `stiva<T>`. Una din implementările posibile este următoarea.

```
template <class T>
class coada: private tablou<T> {
public:
  coada( int d ): tablou<T>( d )
    { head = tail = 0; }
```

```
    ins_q( const T& );
    del_q(       T& );

  private:
    int head; // indicele ultimei locatii ocupate
    int tail; // indicele locatiei predecesoare primei
              // locatii ocupate
  };

  template <class T>
  coada<T>::ins_q( const T& x ) {
    int h = ( head + 1 ) % d;
    if ( h == tail )   return 0;
    a[ head = h ] = x; return 1;
  }

  template <class T>
  coada<T>::del_q( T& x ) {
    if ( head == tail ) return 0;
    tail = ( tail + 1 ) % d;
    x = a[ tail ];      return 1;
  }
```

4.5 Testați funcționarea claselor `stiva<T>` și `coada<T>`, folosind elemente de tip `int`.

Soluție: Dacă programul următor furnizează rezultate corecte, atunci putem avea certitudinea că cele două clase sunt corect implementate.

```
  #include <iostream.h>
  #include "stiva.h"
  #include "coada.h"

  void main( ) {
    int n, i = 0;
    cout << "Numarul elementelor ... "; cin >> n;

    stiva<int> st( n );
    coada<int> cd( n );

    cout << "\nStiva push ... ";
    while ( st.push( i ) )  cout << i++ << ' ';

    cout << "\nStiva pop  ... ";
    while ( st.pop( i ) )   cout << i   << ' ';

    cout << "\nCoada ins_q... ";
    while ( cd.ins_q( i ) ) cout << i++ << ' ';
```

```
  cout << "\nCoada del_q... ";
  while ( cd.del_q( i ) ) cout << i   << ' ';

  cout << '\n';
}
```

4.6 Testați funcționarea clasei parametrice `lista<E>` cu noduri de tip adrese de tablou și apoi cu noduri de tip `tablou<T>`.

Soluție (incorectă): Programul următor nu funcționează corect decât după ce a fost modificat pentru noduri de tip `tablou<T>`. Pentru a-l corecta, nu uitați că toate variabilele din ciclul `for` sunt locale.

```
#include <iostream.h>

#include "tablou.h"
#include "lista.h"

typedef tablou<int> *PTI;

main( ) {
  lista<PTI> tablist;

  for ( int n = 0, i = 0; i < 4; i++ )
  { tablou<int> t( i + 1 );
    for ( int j = t.size( ); j--; t[ j ] = n++ );
    cout << "tablou " << i << ' '; cout << t << '\n';
    tablist.insert( &t );
  }

  cout << "\nLista "; cout << tablist << "\n";

  PTI t; iterator<PTI> it = tablist;
  while( it( t ) )
    cout << "Tablou din lista" << *t << '\n';

  return 1;
}
```

4.7 Destructorul clasei `lista<T>` "distruge" nodurile, invocând procedura iterativă `reset()`. Implementați un destructor în variantă recursivă.

Indicație: Dacă fiecare element de tip `nod<E>` are un destructor de forma `~nod( ) { delete next; }`, atunci destructorul clasei `lista<E>` poate fi `~lista( ) { delete head; }`.

5. Analiza eficienței algoritmilor

Vom dezvolta în acest capitol aparatul matematic necesar pentru analiza eficienței algoritmilor, încercând ca această incursiune matematică să nu fie excesiv de formală. Apoi, vom arăta, pe baza unor exemple, cum poate fi analizat un algoritm. O atenție specială o vom acorda tehnicilor de analiză a algoritmilor recursivi.

5.1 Notația asimptotică

În Capitolul 1 am dat un înțeles intuitiv situației când un algoritm necesită un timp *în ordinul* unei anumite funcții. Revenim acum cu o definiție riguroasă.

5.1.1 O notație pentru "ordinul lui"

Fie **N** mulțimea numerelor naturale (pozitive sau zero) și **R** mulțimea numerelor reale. Notăm prin $\mathbf{N}^+$ și $\mathbf{R}^+$ mulțimea numerelor naturale, respectiv reale, strict pozitive, și prin $\mathbf{R}^*$ mulțimea numerelor reale nenegative. Mulțimea {*true*, *false*} de constante booleene o notăm cu **B**. Fie $f : \mathbf{N} \to \mathbf{R}^*$ o funcție arbitrară. Definim mulțimea

$$O(f) = \{t : \mathbf{N} \to \mathbf{R}^* \mid (\exists c \in \mathbf{R}^+)\ (\exists n_0 \in \mathbf{N})\ (\forall n \geq n_0)\ [t(n) \leq cf(n)]\}$$

Cu alte cuvinte, $O(f)$ (se citește "ordinul lui f") este mulțimea tuturor funcțiilor t mărginite superior de un multiplu real pozitiv al lui f, pentru valori suficient de mari ale argumentului. Vom conveni să spunem că t este în ordinul lui f (sau, echivalent, t este în $O(f)$, sau $t \in O(f)$) chiar și atunci când valoarea $f(n)$ este negativă sau nedefinită pentru anumite valori $n < n_0$. În mod similar, vom vorbi despre ordinul lui f chiar și atunci când valoarea $t(n)$ este negativă sau nedefinită pentru un număr finit de valori ale lui n; în acest caz, vom alege n_0 suficient de mare, astfel încât, pentru $n \geq n_0$, acest lucru să nu mai apară. De exemplu, vom vorbi despre ordinul lui $n/\log n$, chiar dacă pentru $n = 0$ și $n = 1$ funcția nu este definită. În loc de $t \in O(f)$, uneori este mai convenabil să folosim notația $t(n) \in O(f(n))$, subînțelegând aici că $t(n)$ și $f(n)$ sunt funcții.

Fie un algoritm dat și fie o funcție $t : \mathbf{N} \to \mathbf{R}^*$ astfel încât o anumită implementare a algoritmului să necesite cel mult $t(n)$ unități de timp pentru a rezolva un caz de mărime n, $n \in \mathbf{N}$. Principiul invarianței (menționat în Capitolul 1) ne asigură că orice implementare a algoritmului necesită un timp în ordinul lui t. Mai mult, acest algoritm necesită un timp în ordinul lui f pentru orice funcție $f : \mathbf{N} \to \mathbf{R}^*$ pentru care $t \in O(f)$. În particular, $t \in O(t)$. Vom căuta în general să găsim cea mai simplă funcție f, astfel încât $t \in O(f)$.

Proprietățile de bază ale lui $O(f)$ sunt date ca exerciții (Exercițiile 5.1–5.7) și este recomandabil să le studiați înainte de a trece mai departe.

Notația asimptotică definește o relație de ordine parțială între funcții și deci, între eficiența relativă a diferiților algoritmi care rezolvă o anumită problemă. Vom da în continuare o interpretare algebrică a notației asimptotice. Pentru oricare două funcții $f, g : \mathbf{N} \to \mathbf{R}^*$, definim următoarea relație binară: $f \le g$ dacă $O(f) \subseteq O(g)$. Relația "$\le$" este o *relație de ordine parțială* în mulțimea funcțiilor definite pe $\mathbf{N}$ și cu valori în $\mathbf{R}^*$ (Exercițiul 5.6). Definim și o *relație de echivalență*: $f \equiv g$ dacă $O(f) = O(g)$.

În mulțimea $O(f)$ putem înlocui pe f cu orice altă funcție echivalentă cu f. De exemplu, $\lg n \equiv \ln n \equiv \log n$ și avem $O(\lg n) = O(\ln n) = O(\log n)$. Notând cu $O(1)$ ordinul funcțiilor mărginite superior de o constantă, obținem ierarhia:

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset O(2^n)$$

Această ierarhie corespunde unei clasificări a algoritmilor după un criteriu al performanței. Pentru o problemă dată, dorim mereu să obținem un algoritm corespunzător unui ordin cât mai "la stânga". Astfel, este o mare realizare dacă în locul unui algoritm exponențial găsim un algoritm polinomial.

În Exercițiul 5.7 este dată o metodă de simplificare a calculelor, în care apare notația asimptotică. De exemplu,

$$n^3+3n^2+n+8 \in O(n^3+(3n^2+n+8)) = O(\max(n^3, 3n^2+n+8)) = O(n^3)$$

Ultima egalitate este adevărată, chiar dacă $\max(n^3, 3n^2+n+8) \ne n^3$ pentru $0 \le n \le 3$, deoarece notația asimptotică se aplică doar pentru n suficient de mare. De asemenea,

$$\begin{aligned} n^3-3n^2-n-8 \in O(n^3/2+(n^3/2-3n^2-n-8)) &= O(\max(n^3/2, n^3/2-3n^2-n-8)) \\ &= O(n^3/2) = O(n^3) \end{aligned}$$

chiar dacă pentru $0 \le n \le 6$ polinomul este negativ. Exercițiul 5.8 tratează cazul unui polinom oarecare.

Notația $O(f)$ este folosită pentru a limita superior timpul necesar unui algoritm, măsurând eficiența algoritmului respectiv. Uneori este util să estimăm și o limită inferioară a acestui timp. În acest scop, definim mulțimea

$$\Omega(f) = \{t : \mathbf{N} \to \mathbf{R}^* \mid (\exists c \in \mathbf{R}^+)\ (\exists n_0 \in \mathbf{N})\ (\forall n \geq n_0)\ [t(n) \geq cf(n)]\}$$

Există o anumită dualitate între notațiile $O(f)$ și $\Omega(f)$. Și anume, pentru două funcții oarecare $f, g : \mathbf{N} \to \mathbf{R}^*$, avem: $f \in O(g)$, dacă și numai dacă $g \in \Omega(f)$.

O situație fericită este atunci când timpul de execuție al unui algoritm este limitat, atât inferior cât și superior, de câte un multiplu real pozitiv al aceleiași funcții. Introducem notația

$$\Theta(f) = O(f) \cap \Omega(f)$$

numită *ordinul exact* al lui f. Pentru a compara ordinele a două funcții, notația Θ nu este însă mai puternică decât notația O, în sensul că relația $O(f) = O(g)$ este echivalentă cu $\Theta(f) = \Theta(g)$.

Se poate întâmpla ca timpul de execuție al unui algoritm să depindă simultan de mai mulți parametri. Această situație este tipică pentru anumiți algoritmi care operează cu grafuri și în care timpul depinde atât de numărul de vârfuri, cât și de numărul de muchii. Notația asimptotică se generalizează în mod natural și pentru funcții cu mai multe variabile. Astfel, pentru o funcție arbitrară $f : \mathbf{N} \times \mathbf{N} \to \mathbf{R}^*$ definim

$$O(f) = \{t : \mathbf{N} \times \mathbf{N} \to \mathbf{R}^* \mid (\exists c \in \mathbf{R}^+)\ (\exists m_0, n_0 \in \mathbf{N})\ (\forall m \geq m_0)\ (\forall n \geq n_0)\ [t(m, n) \leq cf(m, n)]\}$$

Similar, se obțin și celelalte generalizări.

5.1.2 Notația asimptotică condiționată

Mulți algoritmi sunt mai ușor de analizat dacă considerăm inițial cazuri a căror mărime satisface anumite condiții, de exemplu să fie puteri ale lui 2. În astfel de situații, folosim *notația asimptotică condiționată*. Fie $f : \mathbf{N} \to \mathbf{R}^*$ o funcție arbitrară și fie $P : \mathbf{N} \to \mathbf{B}$ un predicat.

$$O(f \mid P) = \{t : \mathbf{N} \to \mathbf{R}^* \mid (\exists c \in \mathbf{R}^+)\ (\exists n_0 \in \mathbf{N})\ (\forall n \geq n_0)\ [P(n) \Rightarrow t(n) \leq cf(n)]\}$$

Notația $O(f)$ este echivalentă cu $O(f \mid P)$, unde P este predicatul a cărui valoare este mereu *true*. Similar, se obțin notațiile $\Omega(f \mid P)$ și $\Theta(f \mid P)$.

O funcție $f : \mathbf{N} \to \mathbf{R}^*$ este *eventual nedescrescătoare*, dacă există un n_0, astfel încât pentru orice $n \geq n_0$ avem $f(n) \leq f(n+1)$, ceea ce implică prin inducție că, pentru orice $n \geq n_0$ și orice $m \geq n$, avem $f(n) \leq f(m)$. Fie $b \geq 2$ un întreg oarecare. O funcție eventual nedescrescătoare este *b-netedă* dacă $f(bn) \in O(f(n))$. Orice funcție care este b-netedă pentru un anumit $b \geq 2$ este, de asemenea, b-netedă pentru orice $b \geq 2$ (demonstrați acest lucru!); din această cauză, vom spune pur și simplu că aceste funcții sunt *netede*. Următoarea proprietate asamblează aceste definiții, demonstrarea ei fiind lăsată ca exercițiu.

Proprietatea 5.1 Fie $b \geq 2$ un întreg oarecare, $f : \mathbf{N} \to \mathbf{R}^*$ o funcție netedă și $t : \mathbf{N} \to \mathbf{R}^*$ o funcție eventual nedescrescătoare, astfel încât

$$t(n) \in X(f(n) \mid n \text{ este o putere a lui } b)$$

unde X poate fi O, Ω, sau Θ. Atunci, $t \in X(f)$. Mai mult, dacă $t \in \Theta(f)$, atunci și funcția t este netedă. ■

Pentru a înțelege utilitatea notației asimptotice condiționate, să presupunem că timpul de execuție al unui algoritm este dat de ecuația

$$t(n) = \begin{cases} a & \text{pentru } n = 1 \\ t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + bn & \text{pentru } n \neq 1 \end{cases}$$

unde $a, b \in \mathbf{R}^+$ sunt constante arbitrare. Este dificil să analizăm direct această ecuație. Dacă considerăm doar cazurile când n este o putere a lui 2, ecuația devine

$$t(n) = \begin{cases} a & \text{pentru } n = 1 \\ 2t(n/2) + bn & \text{pentru } n > 1 \text{ o putere a lui 2} \end{cases}$$

Prin tehnicile pe care le vom învăța la sfârșitul acestui capitol, ajungem la relația

$$t(n) \in \Theta(n \log n \mid n \text{ este o putere a lui 2})$$

Pentru a arăta acum că $t \in \Theta(n \log n)$, mai trebuie doar să verificăm dacă t este eventual nedescrescătoare și dacă $n \log n$ este netedă.

Prin inducție, vom demonstra că $(\forall n \geq 1)\ [t(n) \leq t(n+1)]$. Pentru început, să notăm că

$$t(1) = a \leq 2(a+b) = t(2)$$

Fie $n > 1$. Presupunem că pentru orice $m < n$ avem $t(m) \leq t(m+1)$. În particular,

$$t(\lfloor n/2 \rfloor) \leq t(\lfloor (n+1)/2 \rfloor)$$
$$t(\lceil n/2 \rceil) \leq t(\lceil (n+1)/2 \rceil)$$

Atunci,

$$t(n) = t(\lfloor n/2 \rfloor)+t(\lceil n/2 \rceil)+bn \le t(\lfloor (n+1)/2 \rfloor)+t(\lceil (n+1)/2 \rceil)+b(n+1) = t(n+1)$$

În fine, mai rămâne să arătăm că $n \log n$ este netedă. Funcția $n \log n$ este eventual nedescrescătoare și

$$\begin{aligned} 2n \log(2n) &= 2n(\log 2 + \log n) = (2 \log 2)n + 2n \log n \\ &\in O(n + n \log n) = O(\max(n, n \log n)) = O(n \log n) \end{aligned}$$

De multe ori, timpul de execuție al unui algoritm se exprimă sub forma unor inegalități de forma

$$t(n) \le \begin{cases} t_1(n) & \text{pentru } n \le n_0 \\ t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + cn & \text{pentru } n > n_0 \end{cases}$$

și, simultan

$$t(n) \ge \begin{cases} t_2(n) & \text{pentru } n \le n_0 \\ t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + dn & \text{pentru } n > n_0 \end{cases}$$

pentru anumite constante $c, d \in \mathbf{R}^+$, $n_0 \in \mathbf{N}$ și pentru două funcții $t_1, t_2 : \mathbf{N} \to \mathbf{R}^+$. Notația asimptotică ne permite să scriem cele două inegalități astfel:

$$t(n) \in t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + O(n)$$

respectiv

$$t(n) \in t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + \Omega(n)$$

Aceste două expresii pot fi scrise și concentrat:

$$t(n) \in t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + \Theta(n)$$

Definim funcția

$$f(n) = \begin{cases} 1 & \text{pentru } n = 1 \\ f(\lfloor n/2 \rfloor) + f(\lceil n/2 \rceil) + n & \text{pentru } n \ne 1 \end{cases}$$

Am văzut că $f \in \Theta(n \log n)$. Ne întoarcem acum la funcția t care satisface inegalitățile precedente. Prin inducție, se demonstrează că există constantele $v \le d$, $u \ge c$, astfel încât

$$v \le t(n)/f(n) \le u$$

pentru orice $n \in \mathbf{N}^+$. Deducem atunci

$$t \in \Theta(f) = \Theta(n \log n)$$

Această tehnică de rezolvare a inegalităților inițiale are două avantaje. În primul rând, nu trebuie să demonstrăm independent că $t \in O(n \log n)$ și $t \in \Omega(n \log n)$. Apoi, mai important, ne permite să restrângem analiza la situația când n este o putere a lui 2, aplicând apoi Proprietatea 5.1. Deoarece nu știm dacă t este eventual nedescrescătoare, nu putem aplica Proprietatea 5.1 direct asupra inegalităților inițiale.

5.2 Tehnici de analiză a algoritmilor

Nu există o formulă generală pentru analiza eficienței unui algoritm. Este mai curând o chestiune de raționament, intuiție și experiență. Vom arăta, pe baza exemplelor, cum se poate efectua o astfel de analiză.

5.2.1 Sortarea prin selecție

Considerăm algoritmul *select* din Secțiunea 1.3. Timpul pentru o singură execuție a buclei interioare poate fi mărginit superior de o constantă a. În total, pentru un i dat, bucla interioară necesită un timp de cel mult $b+a(n-i)$ unități, unde b este o constantă reprezentând timpul necesar pentru inițializarea buclei. O singură execuție a buclei exterioare are loc în cel mult $c+b+a(n-i)$ unități de timp, unde c este o altă constantă. Algoritmul durează în total cel mult

$$d + \sum_{i=1}^{n-1} (c + b + a(n - i))$$

unități de timp, d fiind din nou o constantă. Simplificăm această expresie și obținem

$$(a/2)n^2 + (b+c-a/2)n + (d-c-b)$$

de unde deducem că algoritmul necesită un timp în $O(n^2)$. O analiză similară asupra limitei inferioare arată că timpul este de fapt în $\Theta(n^2)$. Nu este necesar să considerăm cazul cel mai nefavorabil sau cazul mediu, deoarece timpul de execuție este independent de ordonarea prealabilă a elementelor de sortat.

În acest prim exemplu am dat toate detaliile. De obicei, detalii ca inițializarea buclei nu se vor considera explicit. Pentru cele mai multe situații, este suficient să alegem ca *barometru* o anumită instrucțiune din algoritm și să numărăm de câte ori se execută această instrucțiune. În cazul nostru, putem alege ca barometru testul din bucla interioară, acest test executându-se de $n(n-1)/2$ ori. Exercițiul 5.23 ne sugerează că astfel de simplificări trebuie făcute cu discernământ.

5.2.2 Sortarea prin inserție

Timpul pentru algoritmul *insert* (Secțiunea 1.3) este dependent de ordonarea prealabilă a elementelor de sortat. Vom folosi comparația "$x < T[j]$" ca barometru.

Să presupunem că i este fixat și fie $x = T[i]$, ca în algoritm. Cel mai nefavorabil caz apare atunci când $x < T[j]$ pentru fiecare j între 1 și $i-1$, algoritmul făcând în această situație $i-1$ comparații. Acest lucru se întâmplă pentru fiecare valoare a lui i de la 2 la n, atunci când tabloul T este inițial ordonat descrescător. Numărul total de comparații pentru cazul cel mai nefavorabil este

$$\sum_{i=1}^{n}(i-1) = n(n-1)/2 \in \Theta(n^2)$$

Vom estima acum timpul mediu necesar pentru un caz oarecare. Presupunem că elementele tabloului T sunt distincte și că orice permutare a lor are aceeași probabilitate de apariție. Atunci, dacă $1 \le k \le i$, probabilitatea ca $T[i]$ să fie cel de-al k-lea cel mai mare element dintre elementele $T[1], T[2], \ldots, T[i]$ este $1/i$. Pentru un i fixat, condiția $T[i] < T[i-1]$ este falsă cu probabilitatea $1/i$, deci probabilitatea ca să se execute comparația "$x < T[j]$", o singură dată înainte de ieșirea din bucla **while**, este $1/i$. Comparația "$x < T[j]$" se execută de exact două ori tot cu probabilitatea $1/i$ etc. Probabilitatea ca să se execute comparația de exact $i-1$ ori este $2/i$, deoarece aceasta se întâmplă atât când $x < T[1]$, cât și când $T[1] \le x < T[2]$. Pentru un i fixat, numărul mediu de comparații este

$$c_i = 1\cdot 1/i + 2\cdot 1/i + \ldots + (i-2)\cdot 1/i + (i-1)\cdot 2/i = (i+1)/2 - 1/i$$

Pentru a sorta n elemente, avem nevoie de $\sum_{i=2}^{n} c_i$ comparații, ceea ce este egal cu

$$(n^2+3n)/4 - H_n \in \Theta(n^2)$$

unde prin $H_n = \sum_{i=1}^{n} i^{-1} \in \Theta(\log n)$ am notat al n-lea element al seriei armonice (Exercițiul 5.17).

Se observă că algoritmul *insert* efectuează pentru cazul mediu de două ori mai puține comparații decât pentru cazul cel mai nefavorabil. Totuși, în ambele situații, numărul comparațiilor este în $\Theta(n^2)$.

Algoritmul necesită un timp în $\Omega(n^2)$, atât pentru cazul mediu, cât și pentru cel mai nefavorabil. Cu toate acestea, pentru cazul cel mai favorabil, când inițial tabloul este ordonat crescător, timpul este în $O(n)$. De fapt, în acest caz, timpul este și în $\Omega(n)$, deci este în $\Theta(n)$.

5.2.3 Heapsort

Vom analiza, pentru început, algoritmul *make-heap* din Secțiunea 3.4. Definim ca barometru instrucțiunile din bucla **repeat** a algoritmului *sift-down*. Fie m numărul maxim de repetări al acestei bucle, cauzat de apelul lui *sift-down*(T, i), unde i este fixat. Notăm cu j_t valoarea lui j după ce se execută atribuirea "$j \leftarrow k$" la a t-a repetare a buclei. Evident, $j_1 = i$. Dacă $1 < t \le m$, la sfârșitul celei de-a $(t-1)$-a repetări a buclei, avem $j \ne k$ și $k \ge 2j$. În general, $j_t \ge 2j_{t-1}$ pentru $1 < t \le m$. Atunci,

$$n \ge j_m \ge 2j_{m-1} \ge 4j_{m-2} \ge \ldots \ge 2^{m-1}i$$

Rezultă $2^{m-1} \le n/i$, iar de aici obținem relația $m \le 1 + \lg(n/i)$.

Numărul total de executări ale buclei **repeat** la formarea unui heap este mărginit superior de

$$\sum_{i=1}^{a}(1+\lg(n/i)), \text{ unde } a = \lfloor n/2 \rfloor \qquad (*)$$

Pentru a simplifica această expresie, să observăm că pentru orice $k \ge 0$

$$\sum_{i=b}^{c}\lg(n/i) \le 2^k \lg(n/2^k), \text{ unde } b = 2^k \text{ și } c = 2^{k+1}-1$$

Descompunem expresia (*) în secțiuni corespunzătoare puterilor lui 2 și notăm $d = \lfloor \lg(n/2) \rfloor$:

$$\sum_{i=1}^{a}\lg(n/i) \le \sum_{k=0}^{d}2^k \lg(n/2^k) \le 2^{d+1}\lg(n/2^{d-1})$$

Demonstrația ultimei inegalități rezultă din Exercițiul 5.26. Dar $d = \lfloor \lg(n/2) \rfloor$ implică $d+1 \le \lg n$ și $d-1 \ge \lg(n/8)$. Deci,

$$\sum_{i=1}^{a}\lg(n/i) \le 3n$$

Din (*) deducem că $\lfloor n/2 \rfloor + 3n$ repetări ale buclei **repeat** sunt suficiente pentru a construi un heap, deci *make-heap* necesită un timp $t \in O(n)$. Pe de altă parte, deoarece orice algoritm pentru formarea unui heap trebuie să utilizeze fiecare element din tablou cel puțin o dată, $t \in \Omega(n)$. Deci, $t \in \Theta(n)$. Puteți compara acest timp cu timpul necesar algoritmului *slow-make-heap* (Exercițiul 5.28).

Pentru cel mai nefavorabil caz, *sift-down*($T[1 .. i-1]$, 1) necesită un timp în $O(\log n)$ (Exercițiul 5.27). Ținând cont și de faptul că algoritmul *make-heap* este

liniar, rezultă că timpul pentru algoritmul *heapsort* pentru cazul cel mai nefavorabil este în $O(n \log n)$. Mai mult, timpul de execuție pentru *heapsort* este de fapt în $\Theta(n \log n)$, atât pentru cazul cel mai nefavorabil, cât și pentru cazul mediu.

Algoritmii de sortare prezentați până acum au o caracteristică comună: se bazează numai pe comparații între elementele tabloului T. Din această cauză, îi vom numi algoritmi de sortare prin comparație. Vom cunoaște și alți algoritmi de acest tip: *bubblesort*, *quicksort*, *mergesort*. Să observăm că, pentru cel mai nefavorabil caz, orice algoritm de sortare prin comparație necesită un timp în $\Omega(n \log n)$ (Exercițiul 5.30). Pentru cel mai nefavorabil caz, algoritmul *heapsort* este deci optim (în limitele unei constante multiplicative). Același lucru se întâmplă și cu *mergesort*.

5.2.4 Turnurile din Hanoi

Matematicianul francez Éduard Lucas a propus în 1883 o problemă care a devenit apoi celebră, mai ales datorită faptului că a prezentat-o sub forma unei legende. Se spune că Brahma a fixat pe Pământ trei tije de diamant și pe una din ele a pus în ordine crescătoare 64 de discuri de aur de dimensiuni diferite, astfel încât discul cel mai mare era jos. Brahma a creat și o mânăstire, iar sarcina călugărilor era să mute toate discurile pe o altă tijă. Singura operațiune permisă era mutarea a câte unui singur disc de pe o tijă pe alta, astfel încât niciodată să nu se pună un disc mai mare peste unul mai mic. Legenda spune că sfârșitul lumii va fi atunci când călugării vor săvârși lucrarea. Aceasta se dovedește a fi o previziune extrem de optimistă asupra sfârșitului lumii. Presupunând că în fiecare secundă se mută un disc și lucrând fără întrerupere, cele 64 de discuri nu pot fi mutate nici în 500 de miliarde de ani de la începutul acțiunii!

Observăm că pentru a muta cele mai mici n discuri de pe tija i pe tija j (unde $1 \le i \le 3$, $1 \le j \le 3$, $i \ne j$, $n \ge 1$), transferăm cele mai mici $n-1$ discuri de pe tija i pe tija $6-i-j$, apoi transferăm discul n de pe tija i pe tija j, iar apoi retransferăm cele $n-1$ discuri de pe tija $6-i-j$ pe tija j. Cu alte cuvinte, reducem problema mutării a n discuri la problema mutării a $n-1$ discuri. Următoarea procedură descrie acest algoritm recursiv.

```
procedure Hanoi(n, i, j)
    {mută cele mai mici n discuri de pe tija i pe tija j}
    if n > 0 then   Hanoi(n−1, i, 6−i−j)
                    write i "→" j
                    Hanoi(n−1, 6−i−j, j)
```

Pentru rezolvarea problemei inițiale, facem apelul *Hanoi*(64, 1, 2).

Considerăm instrucțiunea **write** ca barometru. Timpul necesar algoritmului este exprimat prin următoarea recurență:

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

Vom demonstra în Secțiunea 5.2 că $t(n) = 2^n - 1$. Rezultă $t \in \Theta(2^n)$.

Acest algoritm este optim, în sensul că este imposibil să mutăm n discuri de pe o tijă pe alta cu mai puțin de $2^n - 1$ operații. Implementarea în oricare limbaj de programare care admite exprimarea recursivă se poate face aproape în mod direct.

5.3 Analiza algoritmilor recursivi

Am văzut în exemplul precedent cât de puternică și, în același timp, cât de elegantă este recursivitatea în elaborarea unui algoritm. Nu vom face o introducere în recursivitate și nici o prezentare a metodelor de eliminare a ei. Cel mai important câștig al exprimării recursive este faptul că ea este naturală și compactă, fără să ascundă esența algoritmului prin detaliile de implementare. Pe de altă parte, apelurile recursive trebuie folosite cu discernământ, deoarece solicită și ele resursele calculatorului (timp și memorie). Analiza unui algoritm recursiv implică rezolvarea unui sistem de recurențe. Vom vedea în continuare cum pot fi rezolvate astfel de recurențe. Începem cu tehnica cea mai banală.

5.3.1 Metoda iterației

Cu puțină experiență și intuiție, putem rezolva de multe ori astfel de recurențe prin *metoda iterației*: se execută primii pași, se intuiește forma generală, iar apoi se demonstrează prin inducție matematică că forma este corectă. Să considerăm de exemplu recurența problemei turnurilor din Hanoi. Pentru un anumit $n > 1$ obținem succesiv

$$t(n) = 2t(n-1) + 1 = 2^2 t(n-2) + 2 + 1 = \ldots = 2^{n-1} t(1) + \sum_{i=0}^{n-2} 2^i$$

Rezultă $t(n) = 2^n - 1$. Prin inducție matematică se demonstrează acum cu ușurință că această formă generală este corectă.

5.3.2 Inducția constructivă

Inducția matematică este folosită de obicei ca tehnică de demonstrare a unei aserțiuni deja enunțate. Vom vedea în această secțiune că inducția matematică poate fi utilizată cu succes și în descoperirea enunțului aserțiunii. Aplicând această tehnică, putem simultan să demonstrăm o aserțiune doar parțial specificată și să descoperim specificațiile care lipsesc și datorită cărora aserțiunea este corectă. Vom vedea că această tehnică a *inducției constructive* este utilă pentru rezolvarea anumitor recurențe care apar în contextul analizei algoritmilor. Începem cu un exemplu.

Fie funcția $f : \mathbf{N} \to \mathbf{N}$, definită prin recurența

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

Să presupunem pentru moment că nu știm că $f(n) = n(n+1)/2$ și să căutăm o astfel de formulă. Avem

$$f(n) = \sum_{i=0}^{n} i \leq \sum_{i=0}^{n} n = n^2$$

și deci, $f(n) \in O(n^2)$. Aceasta ne sugerează să formulăm *ipoteza inducției specificate parțial IISP*(n) conform căreia f este de forma $f(n) = an^2+bn+c$. Această ipoteză este parțială, în sensul că a, b și c nu sunt încă cunoscute. Tehnica inducției constructive constă în a demonstra prin inducție matematică această ipoteză incompletă și a determina în același timp valorile constantelor necunoscute a, b și c.

Presupunem că *IISP*$(n-1)$ este adevărată pentru un anumit $n \geq 1$. Atunci,

$$f(n) = a(n-1)^2+b(n-1)+c+n = an^2+(1+b-2a)n+(a-b+c)$$

Dacă dorim să arătăm că *IISP*(n) este adevărată, trebuie să arătăm că $f(n) = an^2+bn+c$. Prin identificarea coeficienților puterilor lui n, obținem ecuațiile $1+b-2a = b$ și $a-b+c = c$, cu soluția $a = b = 1/2$, c putând fi oarecare. Avem acum o ipoteză mai completă, pe care o numim tot *IISP*(n): $f(n) = n^2/2+n/2+c$. Am arătat că, dacă *IISP*$(n-1)$ este adevărată pentru un anumit $n \geq 1$, atunci este adevărată și *IISP*(n). Rămâne să arătăm că este adevărată și *IISP*(0). Trebuie să arătăm că $f(0) = a\cdot 0+b\cdot 0+c = c$. Știm că $f(0) = 0$, deci *IISP*(0) este adevărată pentru $c = 0$. În concluzie, am demonstrat că $f(n) = n^2/2+n/2$ pentru orice n.

5.3.3 Recurențe liniare omogene

Există, din fericire, și tehnici care pot fi folosite aproape automat pentru a rezolva anumite clase de recurențe. Vom începe prin a considera ecuații recurente liniare omogene, adică de forma

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = 0 \qquad (*)$$

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

Conform intuiției, vom căuta soluții de forma

$$t_n = x^n$$

unde x este o constantă (deocamdată necunoscută). Încercăm această soluție în (*) și obținem

$$a_0 x^n + a_1 x^{n-1} + \ldots + a_k x^{n-k} = 0$$

Soluțiile acestei ecuații sunt fie soluția trivială $x = 0$, care nu ne interesează, fie soluțiile ecuației

$$a_0 x^k + a_1 x^{k-1} + \ldots + a_k = 0$$

care este ecuația caracteristică a recurenței (*).

Presupunând deocamdată că cele k rădăcini $r_1, r_2, \ldots, r_k$ ale acestei ecuații caracteristice sunt distincte, orice combinație liniară

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

este o soluție a recurenței (*), unde constantele $c_1, c_2, \ldots, c_k$ sunt determinate de condițiile inițiale. Este remarcabil că (*) are *numai* soluții de această formă.

Să exemplificăm prin recurența care definește șirul lui Fibonacci (din Secțiunea 1.6.4):

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

iar $t_0 = 0$, $t_1 = 1$. Putem să rescriem această recurență sub forma

$$t_n - t_{n-1} - t_{n-2} = 0$$

care are ecuația caracteristică

$$x^2 - x - 1 = 0$$

cu rădăcinile $r_{1,2} = (1 \pm \sqrt{5})/2$. Soluția generală are forma

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

Impunând condițiile inițiale, obținem

$$\begin{aligned} c_1 + c_2 &= 0 \qquad n = 0 \\ r_1 c_1 + r_2 c_2 &= 1 \qquad n = 1 \end{aligned}$$

de unde determinăm

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

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

$$t_n = 1/\sqrt{5}\,(\phi^n - (-\phi)^{-n})$$

care este cunoscuta relație a lui de Moivre, descoperită la începutul secolului XVI. Nu prezintă nici o dificultate să arătăm acum că timpul pentru algoritmul *fib*1 (din Secțiunea 1.6.4) este în $\Theta(\phi^n)$.

Ce facem însă atunci când rădăcinile ecuației caracteristice nu sunt distincte? Se poate arăta că, dacă r este o rădăcină de multiplicitate m a ecuației caracteristice, atunci $t_n = r^n$, $t_n = nr^n$, $t_n = n^2 r^n$, …, $t_n = n^{m-1} r^n$ sunt soluții pentru (*). Soluția generală pentru o astfel de recurență este atunci o combinație liniară a acestor termeni și a termenilor proveniți de la celelalte rădăcini ale ecuației caracteristice. Din nou, sunt de determinat exact k constante din condițiile inițiale.

Vom da din nou un exemplu. Fie recurența

$$t_n = 5t_{n-1} - 8t_{n-2} + 4t_{n-3} \qquad n \geq 3$$

iar $t_0 = 0$, $t_1 = 1$, $t_2 = 2$. Ecuația caracteristică are rădăcinile 1 (de multiplicitate 1) și 2 (de multiplicitate 2). Soluția generală este:

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

Din condițiile inițiale, obținem $c_1 = -2$, $c_2 = 2$, $c_3 = -1/2$.

5.3.4 Recurențe liniare neomogene

Considerăm acum recurențe de următoarea formă mai generală

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = b^n p(n) \qquad (**)$$

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

De exemplu, o astfel de recurență poate fi:

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

În acest caz, $b = 3$ și $p(n) = 1$, un polinom de grad 0. O simplă manipulare ne permite să reducem acest exemplu la forma (*). Înmulțim recurența cu 3, obținând

$$3t_n - 6t_{n-1} = 3^{n+1}$$

Înlocuind pe n cu $n+1$ în recurența inițială, avem

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

În fine, scădem aceste două ecuații

$$t_{n+1} - 5t_n + 6t_{n-1} = 0$$

Am obținut o recurență omogenă pe care o putem rezolva ca în secțiunea precedentă. Ecuația caracteristică este:

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

adică $(x-2)(x-3) = 0$.

Intuitiv, observăm că factorul $(x-2)$ corespunde părții stângi a recurenței inițiale, în timp ce factorul $(x-3)$ a apărut ca rezultat al manipulărilor efectuate, pentru a scăpa de parte dreaptă.

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

$$(a_0 x^k + a_1 x^{k-1} + \ldots + a_k)(x-b)^{d+1} = 0$$

Odată ce s-a obținut această ecuație, se procedează ca în cazul omogen.

Vom rezolva acum recurența corespunzătoare problemei turnurilor din Hanoi:

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

iar $t_0 = 0$. Rescriem recurența astfel

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

care este de forma (**) cu $b = 1$ și $p(n) = 1$, un polinom de grad 0. Ecuația caracteristică este atunci $(x-2)(x-1) = 0$, cu soluțiile 1 și 2. Soluția generală a recurenței este:

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

Avem nevoie de două condiții inițiale. Știm că $t_0 = 0$; pentru a găsi cea de-a doua condiție calculăm

$$t_1 = 2t_0 + 1$$

Din condițiile inițiale, obținem

$$t_n = 2^n - 1$$

Dacă ne interesează doar ordinul lui t_n, nu este necesar să calculăm efectiv constantele în soluția generală. Dacă știm că $t_n = c_1 1^n + c_2 2^n$, rezultă $t_n \in O(2^n)$. Din faptul că numărul de mutări a unor discuri nu poate fi negativ sau constant, deoarece avem în mod evident $t_n \geq n$, deducem că $c_2 > 0$. Avem atunci $t_n \in \Omega(2^n)$ și deci, $t_n \in \Theta(2^n)$. Putem obține chiar ceva mai mult. Substituind soluția generală înapoi în recurența inițială, găsim

$$1 = t_n - 2t_{n-1} = c_1 + c_2 2^n - 2(c_1 + c_2 2^{n-1}) = -c_1$$

Indiferent de condiția inițială, c_1 este deci -1.

5.3.5 Schimbarea variabilei

Uneori, printr-o schimbare de variabilă, putem rezolva recurențe mult mai complicate. În exemplele care urmează, vom nota cu $T(n)$ termenul general al recurenței și cu t_k termenul noii recurențe obținute printr-o schimbare de variabilă. Presupunem pentru început că n este o putere a lui 2.

Un prim exemplu este recurența

$$T(n) = 4T(n/2) + n \qquad n > 1$$

în care înlocuim pe n cu 2^k, notăm $t_k = T(2^k) = T(n)$ și obținem

$$t_k = 4t_{k-1} + 2^k$$

Ecuația caracteristică a acestei recurențe liniare este

$$(x-4)(x-2) = 0$$

și deci, $t_k = c_1 4^k + c_2 2^k$. Înlocuim la loc pe k cu lg n

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

Rezultă

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

Un al doilea exemplu îl reprezintă ecuația

$$T(n) = 4T(n/2) + n^2 \qquad n > 1$$

Procedând la fel, ajungem la recurența

$$t_k = 4t_{k-1} + 4^k$$

cu ecuația caracteristică

$$(x-4)^2 = 0$$

și soluția generală $t_k = c_1 4^2 + c_2 k 4^2$. Atunci,

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

și obținem

$$T(n) \in O(n^2 \log n \mid n \text{ este o putere a lui 2})$$

În fine, să considerăm și exemplul

$$T(n) = 3T(n/2) + cn \qquad n > 1$$

c fiind o constantă. Obținem succesiv

$$T(2^k) = 3T(2^{k-1}) + c2^k$$

$$t_k = 3t_{k-1} + c2^k$$

cu ecuația caracteristică

$$(x-3)(x-2) = 0$$

$$t_k = c_1 3^k + c_2 2^k$$

$$T(n) = c_1 3^{\lg n} + c_2 n$$

și, deoarece

$$a^{\lg b} = b^{\lg a}$$

obținem

$$T(n) = c_1 n^{\lg 3} + c_2 n$$

deci,

$$T(n) \in O(n^{\lg 3} \mid n \text{ este o putere a lui } 2)$$

În toate aceste exemple am folosit notația asimptotică condiționată. Pentru a arăta că rezultatele obținute sunt adevărate pentru orice n, este suficient să adăugăm condiția ca $T(n)$ să fie eventual nedescrescătoare. Aceasta, datorită Proprietății 5.1 și a faptului că funcțiile n^2, $n \log n$ și $n^{\lg 3}$ sunt netede.

Putem enunța acum o proprietate care este utilă ca rețetă pentru analiza algoritmilor cu recursivități de forma celor din exemplele precedente. Proprietatea, a cărei demonstrare o lăsăm ca exercițiu, ne va fi foarte utilă la analiza algoritmilor divide et impera din Capitolul 7.

Proprietatea 5.2 Fie $T : \mathbf{N} \to \mathbf{R}^+$ o funcție eventual nedescrescătoare

$$T(n) = aT(n/b) + cn^k \qquad n > n_0$$

unde: $n_0 \geq 1$, $b \geq 2$ și $k \geq 0$ sunt întregi; a și c sunt numere reale pozitive; n/n_0 este o putere a lui b. Atunci avem

$$T(n) \in \begin{cases} \Theta(n^k) & \text{pentru } a < b^k \\ \Theta(n^k \log n) & \text{pentru } a = b^k \\ \Theta(n^{\log_b a}) & \text{pentru } a > b^k \end{cases}$$

■

5.4 Exerciții

5.1 Care din următoarele afirmații sunt adevărate?

i) $n^2 \in O(n^3)$

ii) $n^3 \in O(n^2)$

iii) $2^{n+1} \in O(2^n)$

iv) $(n+1)! \in O(n!)$

v) pentru orice funcție $f : \mathbf{N} \to \mathbf{R}^*$, $f \in O(n) \Rightarrow [f^2 \in O(n^2)]$

vi) pentru orice funcție $f : \mathbf{N} \to \mathbf{R}^*$, $f \in O(n) \Rightarrow [2^f \in O(2^n)]$

5.2 Presupunând că f este strict pozitivă pe $\mathbf{N}$, demonstrați că definiția lui $O(f)$ este echivalentă cu următoarea definiție:

$$O(f) = \{t : \mathbf{N} \to \mathbf{R}^* \mid (\exists c \in \mathbf{R}^+)\ (\forall n \in \mathbf{N})\ [t(n) \leq cf(n)]\}$$

5.3 Demonstrați că relația "$\in O$" este *tranzitivă*: dacă $f \in O(g)$ și $g \in O(h)$, atunci $f \in O(h)$. Deduceți de aici că dacă $g \in O(h)$, atunci $O(g) \subseteq O(h)$.

5.4 Pentru oricare două funcții $f, g : \mathbf{N} \to \mathbf{R}^*$, demonstrați că:

i) $O(f) = O(g) \quad \Leftrightarrow \quad f \in O(g)$ și $g \in O(f)$

ii) $O(f) \subset O(g) \quad \Leftrightarrow \quad f \in O(g)$ și $g \notin O(f)$

5.5 Găsiți două funcții $f, g : \mathbf{N} \to \mathbf{R}^*$, astfel încât $f \notin O(g)$ și $g \notin O(f)$.

Indicație: $f(n) = n$, $g(n) = n^{1+\sin n}$

5.6 Pentru oricare două funcții $f, g : \mathbf{N} \to \mathbf{R}^*$ definim următoarea relație binară: $f \leq g$ dacă $O(f) \subseteq O(g)$. Demonstrați că relația "$\leq$" este o relație de ordine parțială în mulțimea funcțiilor definite pe $\mathbf{N}$ și cu valori în $\mathbf{R}^*$.

Indicație: Trebuie arătat că relația este parțială, reflexivă, tranzitivă și antisimetrică. Țineți cont de Exercițiul 5.5.

5.7 Pentru oricare două funcții $f, g : \mathbf{N} \to \mathbf{R}^*$ demonstrați că

$$O(f + g) = O(\max(f, g))$$

unde suma și maximul se iau punctual.

5.8 Fie $f(n) = a_m n^m + \ldots + a_1 n + a_0$ un polinom de grad m, cu $a_m > 0$. Arătați că $f \in O(n^m)$.

5.9 $O(n^2) = O(n^3 + (n^2 - n^3)) = O(\max(n^3, n^2 - n^3)) = O(n^3)$

Unde este eroarea?

5.10 Găsiți eroarea în următorul lanț de relații:

$$\sum_{i=1}^{n} i = 1+2+\ldots+n \in O(1+2+\ldots+n) = O(\max(1, 2, \ldots, n)) = O(n)$$

5.11 Fie $f, g : \mathbf{N} \to \mathbf{R}^+$. Demonstrați că:

i) $\lim_{n\to\infty} f(n) / g(n) \in \mathbf{R}^+ \quad \Rightarrow \quad O(f) = O(g)$

ii) $\lim_{n\to\infty} f(n) / g(n) = 0 \quad \Rightarrow \quad O(f) \subset O(g)$

Observație: Implicațiile inverse nu sunt în general adevărate, deoarece se poate întâmpla ca limitele să nu existe.

5.12 Folosind regula lui l'Hôspital și Exercițiile 5.4, 5.11, arătați că

$$\log n \in O(\sqrt{n}), \quad \text{dar} \quad \sqrt{n} \notin O(\log n)$$

Indicație: Prelungim domeniile funcțiilor pe $\mathbf{R}^+$, pe care sunt derivabile și aplicăm regula lui l'Hôspital pentru $\log n/\sqrt{n}$.

5.13 Pentru oricare $f, g : \mathbf{N} \to \mathbf{R}^*$, demonstrați că:

$$f \in O(g) \quad \Leftrightarrow \quad g \in \Omega(f)$$

5.14 Arătați că $f \in \Theta(g)$ dacă și numai dacă

$$(\exists c, d \in \mathbf{R}^+)\ (\exists n_0 \in \mathbf{N})\ (\forall n \geq n_0)\ [cg(n) \leq f(n) \leq dg(n)]$$

5.15 Demonstrați că următoarele propoziții sunt echivalente, pentru oricare două funcții $f, g : \mathbf{N} \to \mathbf{R}^*$.

i) $O(f) = O(g)$

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

iii) $f \in \Theta(g)$

5.16 Continuând Exercițiul 5.11, arătați că pentru oricare două funcții $f, g : \mathbf{N} \to \mathbf{R}^+$ avem:

i) $\lim_{n\to\infty} f(n) / g(n) \in \mathbf{R}^+ \quad \Rightarrow \quad f \in \Theta(g)$

ii) $\lim_{n\to\infty} f(n) / g(n) = 0 \quad \Rightarrow \quad f \in O(g) \quad \text{dar} \quad f \notin \Theta(g)$

iii) $\lim_{n\to\infty} f(n) / g(n) = +\infty \quad \Rightarrow \quad f \in \Omega(g)$ dar $f \notin \Theta(g)$

5.17 Demonstrați următoarele afirmații:

i) $\log_a n \in \Theta(\log_b n)$ pentru oricare $a, b > 1$

ii) $\sum_{i=1}^{n} i^k \in \Theta(n^{k+1})$ pentru oricare $k \in \mathbf{N}$

iii) $\sum_{i=1}^{n} i^{-1} \in \Theta(\log n)$

iv) $\log n! \in \Theta(n \log n)$

Indicație: La punctul *iii)* se ține cont de relația:

$$\sum_{i=1}^{\infty} i^{-1} = \ln n + \gamma + 1/2n - 1/12n^2 + \ldots$$

unde $\gamma = 0{,}5772\ldots$ este constanta lui Euler.

La punctul *iv)*, din $n! < n^n$, rezultă $\log n! \in O(n \log n)$. Să arătăm acum, că $\log n! \in \Omega(n \log n)$. Pentru $0 \le i \le n-1$ este adevărată relația

$$(n-i)(i+1) \ge n$$

Deoarece

$$(n!)^2 = (n\cdot 1)\,((n-1)\cdot 2)\,((n-2)\cdot 3)\cdot\ldots\cdot(2\cdot(n-1))\,(1\cdot n) \ge n^n$$

rezultă $2 \log n! \ge n \log n$ și deci $\log n! \in \Omega(n \log n)$.

Punctul *iv)* se poate demonstra și altfel, considerând aproximarea lui Stirling:

$$n! \in \sqrt{2\pi n}\ (n/e)^n (1+\Theta(1/n))$$

unde $e = 1{,}71828\ldots$.

5.18 Arătați că timpul de execuție al unui algoritm este în $\Theta(g)$, $g : \mathbf{N} \to \mathbf{R}^*$, dacă și numai dacă: timpul este în $O(g)$ pentru cazul cel mai nefavorabil și în $\Omega(g)$ pentru cazul cel mai favorabil.

5.19 Pentru oricare două funcții $f, g : \mathbf{N} \to \mathbf{R}^*$ demonstrați că

$$\Theta(f) + \Theta(g) = \Theta(f + g) = \Theta(\max(f, g)) = \max(\Theta(f), \Theta(g))$$

unde suma și maximul se iau punctual.

5.20 Demonstrați Proprietatea 5.1. Arătați pe baza unor contraexemple că cele două condiții "$t(n)$ este eventual nedescrescătoare" și "$f(bn) \in O(f(n))$" sunt necesare.

5.21 Analizați eficiența următorilor patru algoritmi:

```
for i ← 1 to n do
    for j ←1 to 5 do
        {operație elementară}
```

```
for i ← 1 to n do
    for j ← 1 to i+1 do
        {operație elementară}
```

```
for i ← 1 to n do
    for j ← 1 to 6 do
        for k ←1 to n do
            {operație elementară}
```

```
for i ← 1 to n do
    for j ← 1 to i do
        for k ← 1 to n do
            {operație elementară}
```

5.22 Construiți un algoritm cu timpul în $\Theta(n \log n)$.

5.23 Fie următorul algoritm

```
k ← 0
  for i ← 1 to n do
      for j ← 1 to T[i] do
          k ← k+T[ j]
```

unde T este un tablou de n întregi nenegativi. În ce ordin este timpul de execuție al algoritmului?

Soluție: Fie s suma elementelor lui T. Dacă alegem ca barometru instrucțiunea "$k \leftarrow k+T[\,j]$", calculăm că ea se execută de s ori. Deci, am putea deduce că timpul este în ordinul exact al lui s. Un exemplu simplu ne va convinge că am greșit. Presupunem că $T[i] = 1$, atunci când i este un pătrat perfect, și $T[i] = 0$, în rest. În acest caz, $s = \lfloor \sqrt{n} \rfloor$. Totuși, algoritmul necesită timp în ordinul lui $\Omega(n)$, deoarece fiecare element al lui T este considerat cel puțin o dată. Nu am ținut cont de următoarea regulă simplă: putem neglija timpul necesar inițializării și controlului unei bucle, dar cu condiția să includem "ceva" de fiecare dată când se execută bucla.

Iată acum analiza detailată a algoritmului. Fie a timpul necesar pentru o executare a buclei interioare, inclusiv partea de control. Executarea completă a buclei interioare, pentru un i dat, necesită $b+aT[i]$ unități de timp, unde constanta b reprezintă timpul pentru inițializarea buclei. Acest timp nu este zero, când $T[i] = 0$. Timpul pentru o executare a buclei exterioare este $c+b+aT[i]$, c fiind o

nouă constantă. În fine, întregul algoritm necesită $d + \sum_{i=1}^{n}(c + b + aT[i])$ unități de timp, unde d este o altă constantă. Simplificând, obținem $(c+b)n+as+d$. Timpul $t(n, s)$ depinde deci de doi parametri independenți n și s. Avem: $t \in \Theta(n+s)$ sau, ținând cont de Exercițiul 5.19, $t \in \Theta(\max(n, s))$.

5.24 Pentru un tablou $T[1 .. n]$, fie următorul algoritm de sortare:

```
for i ← n downto 1 do
    for j ← 2 to i do
        if T[ j−1] > T[ j] then interschimbă T[ j−1] și T[ j]
```

Această tehnică de sortare se numește *metoda bulelor* (*bubble sort*).

i) Analizați eficiența algoritmului, luând ca barometru testul din bucla interioară.

ii) Modificați algoritmul, astfel încât, dacă pentru un anumit i nu are loc nici o interschimbare, atunci algoritmul se oprește. Analizați eficiența noului algoritm.

5.25 Fie următorul algoritm

```
for i ← 0 to n do
    j ← i
    while j ≠ 0 do j ← j div 2
```

Găsiți ordinul exact al timpului de execuție.

5.26 Demonstrați că pentru oricare întregi pozitivi n și d

$$\sum_{k=0}^{d} 2^k \lg(n / 2^k) = 2^{d+1} \lg(n / 2^{d-1}) - 2 - \lg n$$

Soluție:

$$\sum_{k=0}^{d} 2^k \lg(n / 2^k) = (2^{d+1} - 1) \lg n - \sum_{k=0}^{d} (2^k k)$$

Mai rămâne să arătați că

$$\sum_{k=0}^{d} (2^k k) = (d-1)2^{d+1} + 2$$

5.27 Analizați algoritmii *percolate* și *sift-down* pentru cel mai nefavorabil caz, presupunând că operează asupra unui heap cu n elemente.

Indicație: În cazul cel mai nefavorabil, algoritmii *percolate* și *sift-down* necesită un timp în ordinul exact al înălțimii arborelui complet care reprezintă heap-ul, adică în $\Theta(\lfloor \lg n \rfloor) = \Theta(\log n)$.

5.28 Analizați algoritmul *slow-make-heap* pentru cel mai nefavorabil caz.

Soluție: Pentru *slow-make-heap*, cazul cel mai nefavorabil este atunci când, inițial, T este ordonat crescător. La pasul i, se apelează *percolate*(T[1 .. i], i), care efectuează $\lfloor \lg i \rfloor$ comparații între elemente ale lui T. Numărul total de comparații este atunci

$$C(n) \leq (n-1) \lfloor \lg n \rfloor \in O(n \log n)$$

Pe de altă parte, avem

$$C(n) = \sum_{i=2}^{n} \lfloor \lg i \rfloor > \sum_{i=2}^{n} (\lg i - 1) = \lg n! - (n-1)$$

În Exercițiul 5.17 am arătat că $\lg n! \in \Omega(n \log n)$. Rezultă $C(n) \in \Omega(n \log n)$ și timpul este deci în $\Theta(n \log n)$.

5.29 Arătați că, pentru cel mai nefavorabil caz, timpul de execuție al algoritmului *heapsort* este și în $\Omega(n \log n)$, deci în $\Theta(n \log n)$.

5.30 Demonstrați că, pentru cel mai nefavorabil caz, orice algoritm de sortare prin comparație necesită un timp în $\Omega(n \log n)$. În particular, obținem astfel, pe altă cale, rezultatul din Exercițiul 5.29.

Soluție: Orice sortare prin comparație poate fi interpretată ca o parcurgere a unui arbore binar de decizie, prin care se stabilește ordinea relativă a elementelor de sortat. Într-un arbore binar de decizie, fiecare vârf neterminal semnifică o comparație între două elemente ale tabloului T și fiecare vârf terminal reprezintă o permutare a elementelor lui T. Executarea unui algoritm de sortare corespunde parcurgerii unui drum de la rădăcina arborelui de decizie către un vârf terminal. La fiecare vârf neterminal se efectuează o comparație între două elemente $T[i]$ și $T[j]$: dacă $T[i] \leq T[j]$ se continuă cu comparațiile din subarborele stâng, iar în caz contrar cu cele din subarborele drept. Când se ajunge la un vârf terminal, înseamnă că algoritmul de sortare a reușit să stabilească ordinea elementelor din T.

Fiecare din cele $n!$ permutări a celor n elemente trebuie să apară ca vârf terminal în arborele de decizie. Vom lua ca barometru comparația între două elemente ale tabloului T. Înălțimea h a arborelui de decizie corespunde numărului de comparații pentru cel mai nefavorabil caz. Deoarece căutăm limita inferioară a timpului, ne interesează doar algoritmii cei mai performanți de sortare, deci putem presupune că numărul de vârfuri este minim, adică $n!$. Avem: $n! \leq 2^h$ (demonstrați acest lucru!), adică $h \geq \lg n!$. Considerând și relația $\log n! \in \Omega(n \log n)$ (vezi Exercițiul 5.17), rezultă că timpul de execuție pentru orice algoritm de sortare prin comparație este, în cazul cel mai nefavorabil, în $\Omega(n \log n)$.

5.31 Analizați algoritmul *heapsort* pentru cel mai favorabil caz. Care este cel mai favorabil caz?

5.32 Analizați algoritmii *fib*2 și *fib*3 din Secțiunea 1.6.4.

Soluție:

i) Se deduce imediat că timpul pentru *fib*2 este în $\Theta(n)$.

ii) Pentru a analiza algoritmul *fib*3, luăm ca barometru instrucțiunile din bucla **while**. Fie n_t valoarea lui n la sfârșitul executării celei de-a t-a bucle. În particular, $n_1 = \lfloor n/2 \rfloor$. Dacă $2 \leq t \leq m$, atunci

$$n_t = \lfloor n_{t-1}/2 \rfloor \leq n_{t-1}/2$$

Deci,

$$n_t \leq n_{t-1}/2 \leq \ldots \leq n/2^t$$

Fie $m = 1 + \lfloor \lg n \rfloor$. Deducem:

$$n_m \leq n/2^m < 1$$

Dar, $n_m \in \mathbf{N}$, și deci, $n_m = 0$, care este condiția de ieșire din buclă. Cu alte cuvinte, bucla este executată de cel mult m ori, timpul lui *fib*3 fiind în $O(\log n)$. Arătați că timpul este de fapt în $\Theta(\log n)$.

La analiza acestor doi algoritmi, am presupus implicit că operațiile efectuate sunt independente de mărimea operanzilor. Astfel, timpul necesar adunării a două numere este independent de mărimea numerelor și este mărginit superior de o constantă. Dacă nu mai considerăm această ipoteză, atunci analiza se complică.

5.33 Rezolvați recurența $t_n - 3t_{n-1} - 4t_{n-2} = 0$, unde $n \geq 2$, iar $t_0 = 0$, $t_1 = 1$.

5.34 Care este ordinul timpului de execuție pentru un algoritm recursiv cu recurența $t_n = 2t_{n-1} + n$.

Indicație: Se ajunge la ecuația caracteristică $(x-2)(x-1)^2 = 0$, iar soluția generală este $t_n = c_1 2^n + c_2 1^n + c_3 n 1^n$. Rezultă $t \in O(2^n)$.

Substituind soluția generală înapoi în recurență, obținem că, indiferent de condiția inițială, $c_2 = -2$ și $c_3 = -1$. Atunci, toate soluțiile interesante ale recurenței trebuie să aibă $c_1 > 0$ și ele sunt toate în $\Omega(2^n)$, deci în $\Theta(2^n)$.

5.35 Scrieți o variantă recursivă a algoritmului de sortare prin inserție și determinați ordinul timpului de execuție pentru cel mai nefavorabil caz.

Indicație: Pentru a sorta $T[1 .. n]$, sortăm recursiv $T[1 .. n-1]$ și inserăm $T[n]$ în tabloul sortat $T[1 .. n-1]$.

5.36 Determinați prin schimbare de variabilă ordinul timpului de execuție pentru un algoritm cu recurența $T(n) = 2T(n/2) + n \lg n$, unde $n > 1$ este o putere a lui 2.

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

5.37 Demonstrați Proprietatea 5.2, folosind tehnica schimbării de variabilă.

6. Algoritmi greedy

Puși în fața unei probleme pentru care trebuie să elaborăm un algoritm, de multe ori "nu știm cum să începem". Ca și în orice altă activitate, există câteva principii generale care ne pot ajuta în această situație. Ne propunem să prezentăm în următoarele capitole tehnicile fundamentale de elaborare a algoritmilor. Câteva din aceste metode sunt atât de generale, încât le folosim frecvent, chiar dacă numai intuitiv, ca reguli elementare în gândire.

6.1 Tehnica greedy

Algoritmii *greedy* (greedy = lacom) sunt în general simpli și sunt folosiți la probleme 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. În cele mai multe situații de acest fel avem:

- o mulțime de *candidați* (lucrări de executat, vârfuri ale grafului etc)
- o funcție care verifică dacă o anumită mulțime de candidați constituie o *soluție posibilă*, nu neapărat optimă, a problemei
- o funcție care verifică dacă o mulțime de candidați este *fezabilă*, adică dacă este posibil să completăm această mulțime astfel încât să obținem o soluție posibilă, nu neapărat optimă, a problemei
- o *funcție de selecție* care indică la orice moment care este cel mai promițător dintre candidații încă nefolosiți
- o *funcție obiectiv* care dă valoarea unei soluții (timpul necesar executării tuturor lucrărilor într-o anumită ordine, lungimea drumului pe care l-am găsit etc); aceasta este funcția pe care urmărim să o optimizăm (minimizăm/maximizăm)

Pentru a rezolva problema noastră de optimizare, căutăm o soluție posibilă care să optimizeze valoarea funcției obiectiv. Un algoritm greedy construiește soluția pas cu pas. Inițial, mulțimea candidaților selectați este vidă. La fiecare pas, încercăm să adăugăm acestei mulțimi cel mai promițător candidat, conform funcției de selecție. Dacă, după o astfel de adăugare, mulțimea de candidați selectați nu mai este fezabilă, eliminăm ultimul candidat adăugat; acesta nu va mai fi niciodată considerat. Dacă, după adăugare, mulțimea de candidați selectați este fezabilă, ultimul candidat adăugat va rămâne de acum încolo în ea. De fiecare dată când lărgim mulțimea candidaților selectați, verificăm dacă această mulțime nu constituie o soluție posibilă a problemei noastre. Dacă algoritmul greedy

funcționează corect, prima soluție găsită va fi totodată o soluție optimă a problemei. Soluția optimă nu este în mod necesar unică: se poate ca funcția obiectiv să aibă aceeași valoare optimă pentru mai multe soluții posibile. Descrierea formală a unui algoritm greedy general este:

function *greedy*(*C*)
  {*C* este mulțimea candidaților}
  $S \leftarrow \emptyset$ {*S* este mulțimea în care construim soluția}
  while not *soluție*(*S*) **and** $C \neq \emptyset$ **do**
    $x \leftarrow$ un element din *C* care maximizează/minimizează *select*(*x*)
    $C \leftarrow C \setminus \{x\}$
    if *fezabil*($S \cup \{x\}$) **then** $S \leftarrow S \cup \{x\}$
  if *soluție*(*S*) **then** **return** *S*
  else **return** "nu există soluție"

Este de înțeles acum de ce un astfel de algoritm se numește "lacom" (am putea să-i spunem și "nechibzuit"). La fiecare pas, procedura alege cel mai bun candidat la momentul respectiv, fără să-i pese de viitor și fără să se răzgândească. Dacă un candidat este inclus în soluție, el rămâne acolo; dacă un candidat este exclus din soluție, el nu va mai fi niciodată reconsiderat. Asemenea unui întreprinzător rudimentar care urmărește câștigul imediat în dauna celui de perspectivă, un algoritm greedy acționează simplist. Totuși, ca și în afaceri, o astfel de metodă poate da rezultate foarte bune tocmai datorită simplității ei.

Funcția *select* este de obicei derivată din funcția obiectiv; uneori aceste două funcții sunt chiar identice.

Un exemplu simplu de algoritm greedy este algoritmul folosit pentru rezolvarea următoarei probleme. Să presupunem că dorim să dăm restul unui client, folosind un număr cât mai mic de monezi. În acest caz, elementele problemei sunt:

- candidații: mulțimea inițială de monezi de 1, 5, și 25 unități, în care presupunem că din fiecare tip de monedă avem o cantitate nelimitată
- o soluție posibilă: valoarea totală a unei astfel de mulțimi de monezi selectate trebuie să fie exact valoarea pe care trebuie să o dăm ca rest
- o mulțime fezabilă: valoarea totală a unei astfel de mulțimi de monezi selectate nu este mai mare decât valoarea pe care trebuie să o dăm ca rest
- funcția de selecție: se alege cea mai mare monedă din mulțimea de candidați rămasă
- funcția obiectiv: numărul de monezi folosite în soluție; se dorește minimizarea acestui număr

Se poate demonstra că algoritmul greedy va găsi în acest caz mereu soluția optimă (restul cu un număr minim de monezi). Pe de altă parte, presupunând că există și monezi de 12 unități sau că unele din tipurile de monezi lipsesc din mulțimea

inițială de candidați, se pot găsi contraexemple pentru care algoritmul nu găsește soluția optimă, sau nu găsește nici o soluție cu toate că există soluție.

Evident, soluția optimă se poate găsi încercând toate combinările posibile de monezi. Acest mod de lucru necesită însă foarte mult timp.

Un algoritm greedy nu duce deci întotdeauna la soluția optimă, sau la o soluție. Este doar un principiu general, urmând ca pentru fiecare caz în parte să determinăm dacă obținem sau nu soluția optimă.

6.2 Minimizarea timpului mediu de așteptare

O singură stație de servire (procesor, pompă de benzină etc) trebuie să satisfacă cererile a n clienți. Timpul de servire necesar fiecărui client este cunoscut în prealabil: pentru clientul i este necesar un timp t_i, $1 \le i \le n$. Dorim să minimizăm timpul total de așteptare

$$T = \sum_{i=1}^{n} \text{(timpul de așteptare pentru clientul } i\text{)}$$

ceea ce este același lucru cu a minimiza timpul mediu de așteptare, care este T/n. De exemplu, dacă avem trei clienți cu $t_1 = 5$, $t_2 = 10$, $t_3 = 3$, sunt posibile șase ordini de servire. În primul caz, clientul 1 este servit primul, clientul 2 așteaptă până este servit clientul 1 și apoi este servit, clientul 3 așteaptă până sunt serviți clienții 1, 2 și apoi este servit. Timpul total de așteptare a celor trei clienți este 38.

Ordinea			T			
1	2	3	5+(5+10)+(5+10+3)	=	38	
1	3	2	5+(5+3)+(5+3+10)	=	31	
2	1	3	10+(10+5)+(10+5+3)	=	43	
2	3	1	10+(10+3)+(10+3+5)	=	41	
3	1	2	3+(3+5)+(3+5+10)	=	29	← optim
3	2	1	3+(3+10)+(3+10+5)	=	34	

Algoritmul greedy este foarte simplu: la fiecare pas se selectează clientul cu timpul minim de servire din mulțimea de clienți rămasă. Vom demonstra că acest algoritm este optim. Fie

$$I = (i_1 \ i_2 \ \ldots \ i_n)$$

o permutare oarecare a întregilor $\{1, 2, \ldots, n\}$. Dacă servirea are loc în ordinea I, avem

$$T(I) = t_{i_1} + (t_{i_1} + t_{i_2}) + (t_{i_1} + t_{i_2} + t_{i_3}) + \ldots = n t_{i_1} + (n-1) t_{i_2} + \ldots = \sum_{k=1}^{n} (n-k+1)\, t_{i_k}$$

Presupunem acum că I este astfel încât putem găsi doi întregi $a < b$ cu

$$t_{i_a} > t_{i_b}$$

Interschimbăm pe i_a cu i_b în I; cu alte cuvinte, clientul care a fost servit al b-lea va fi servit acum al a-lea și invers. Obținem o nouă ordine de servire J, care este de preferat deoarece

$$T(J) = (n-a+1)\, t_{i_b} + (n-b+1)\, t_{i_a} + \sum_{\substack{k=1 \\ k \neq a,b}}^{n} (n-k+1)\, t_{i_k}$$

$$T(I) - T(J) = (n-a+1)(t_{i_a} - t_{i_b}) + (n-b+1)(t_{i_b} - t_{i_a}) = (b-a)(t_{i_a} - t_{i_b}) > 0$$

Prin metoda greedy obținem deci întotdeauna planificarea optimă a clienților.

Problema poate fi generalizată pentru un sistem cu mai multe stații de servire.

6.3 Interclasarea optimă a șirurilor ordonate

Să presupunem că avem două șiruri S_1 și S_2 ordonate crescător și că dorim să obținem prin interclasarea lor șirul ordonat crescător care conține elementele din cele două șiruri. Dacă interclasarea are loc prin deplasarea elementelor din cele două șiruri în noul șir rezultat, atunci numărul deplasărilor este $\#S_1 + \#S_2$.

Generalizând, să considerăm acum n șiruri $S_1, S_2, \ldots, S_n$, fiecare șir S_i, $1 \le i \le n$, fiind format din q_i elemente ordonate crescător (vom numi q_i *lungimea* lui S_i). Ne propunem să obținem șirul S ordonat crescător, conținând exact elementele din cele n șiruri. Vom realiza acest lucru prin interclasări succesive de câte două șiruri. Problema constă în determinarea ordinii optime în care trebuie efectuate aceste interclasări, astfel încât numărul total al deplasărilor să fie cât mai mic. Exemplul de mai jos ne arată că problema astfel formulată nu este banală, adică nu este indiferent în ce ordine se fac interclasările.

Fie șirurile S_1, S_2, S_3 de lungimi $q_1 = 30$, $q_2 = 20$, $q_3 = 10$. Dacă interclasăm pe S_1 cu S_2, iar rezultatul îl interclasăm cu S_3, numărul total al deplasărilor este (30+20)+(50+10) = 110. Dacă îl interclasăm pe S_3 cu S_2, iar rezultatul îl interclasăm cu S_1, numărul total al deplasărilor este (10+20)+(30+30) = 90.

Atașăm fiecărei strategii de interclasare câte un arbore binar în care valoarea fiecărui vârf este dată de lungimea șirului pe care îl reprezintă. Dacă șirurile

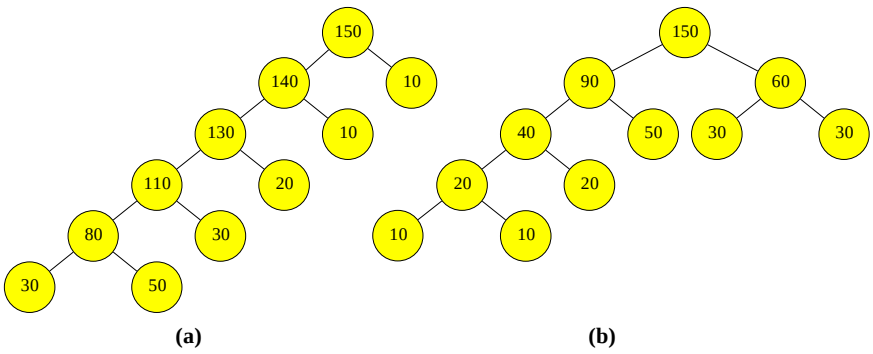


Figura 6.1 Reprezentarea strategiilor de interclasare.

S_1, S_2, …, S_6 au lungimile $q_1 = 30$, $q_2 = 10$, $q_3 = 20$, $q_4 = 30$, $q_5 = 50$, $q_6 = 10$, două astfel de strategii de interclasare sunt reprezentate prin arborii din Figura 6.1.

Observăm că fiecare arbore are 6 vârfuri terminale, corespunzând celor 6 șiruri inițiale și 5 vârfuri neterminale, corespunzând celor 5 interclasări care definesc strategia respectivă. Numerotăm vârfurile în felul următor: vârful terminal i, $1 \leq i \leq 6$, va corespunde șirului S_i, iar vârfurile neterminale se numerotează de la 7 la 11 în ordinea obținerii interclasărilor respective (Figura 6.2).

Strategia greedy apare în Figura 6.1b și constă în a interclasa mereu cele mai scurte două șiruri disponibile la momentul respectiv.

Interclasând șirurile S_1, S_2, …, S_n, de lungimi q_1, q_2, …, q_n, obținem pentru

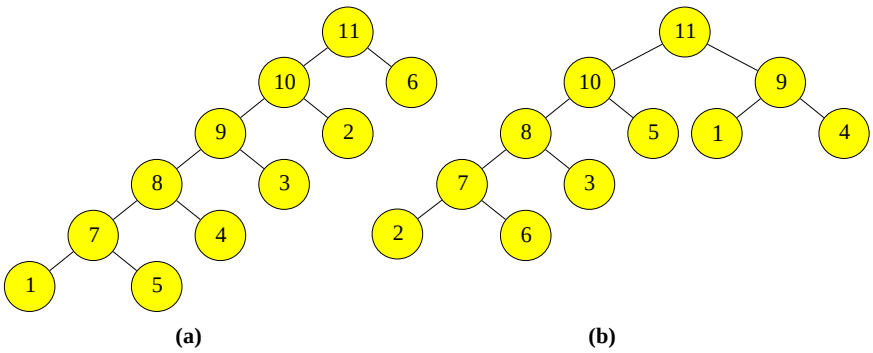


Figura 6.2 Numerotarea vârfurilor arborilor din Figura 6.1.

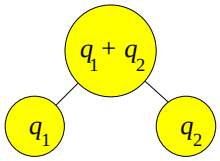
fiecare strategie câte un arbore binar cu n vârfuri terminale, numerotate de la 1 la n, și n–1 vârfuri neterminale, numerotate de la n+1 la 2n–1. Definim, pentru un arbore oarecare A de acest tip, *lungimea externă ponderată*:

$$L(A) = \sum_{i=1}^{n} a_i q_i$$

unde a_i este adâncimea vârfului i. Se observă că numărul total de deplasări de elemente pentru strategia corespunzătoare lui A este chiar $L(A)$. Soluția optimă a problemei noastre este atunci arborele (strategia) pentru care lungimea externă ponderată este minimă.

Proprietatea 6.1 Prin metoda greedy se obține întotdeauna interclasarea optimă a n șiruri ordonate, deci strategia cu arborele de lungime externă ponderată minimă.

Demonstrație: Demonstrăm prin inducție. Pentru $n = 1$, proprietatea este verificată. Presupunem că proprietatea este adevărată pentru n–1 șiruri. Fie A arborele strategiei greedy de interclasare a n șiruri de lungime $q_1 \le q_2 \le \ldots q_n$. Fie B un arbore cu lungimea externă ponderată minimă, corespunzător unei strategii optime de interclasare a celor n șiruri. În arborele A apare subarborele



reprezentând prima interclasare făcută conform strategiei greedy. În arborele B, fie un vârf neterminal de adâncime maximă. Cei doi fii ai acestui vârf sunt atunci două vârfuri terminale q_j și q_k. Fie B' arborele obținut din B schimbând între ele vârfurile q_1 și q_j, respectiv q_2 și q_k. Evident, $L(B') \le L(B)$. Deoarece B are lungimea externă ponderată minimă, rezultă că $L(B') = L(B)$. Eliminând din B' vârfurile q_1 și q_2, obținem un arbore B'' cu n–1 vârfuri terminale q_1+q_2, q_3, …, q_n. Arborele B' are lungimea externă ponderată minimă și $L(B') = L(B'') + (q_1+q_2)$. Rezultă că și B'' are lungimea externă ponderată minimă. Atunci, conform ipotezei inducției, avem $L(B'') = L(A')$, unde A' este arborele strategiei greedy de interclasare a șirurilor de lungime q_1+q_2, q_3, …, q_n. Cum A se obține din A' atașând la vârful q_1+q_2 fiii q_1 și q_2, iar B' se obține în același mod din B'', rezultă că $L(A) = L(B') = L(B)$. Proprietatea este deci adevărată pentru orice n. ■

La scrierea algoritmului care generează arborele strategiei greedy de interclasare

vom folosi un min-heap. Fiecare element al min-heap-ului este o pereche (q, i) unde i este numărul unui vârf din arborele strategiei de interclasare, iar q este lungimea șirului pe care îl reprezintă. Proprietatea de min-heap se referă la valoarea lui q.

Algoritmul *interopt* va construi arborele strategiei greedy. Un vârf i al arborelui va fi memorat în trei locații diferite conținând:

$LU[i]$ = lungimea șirului reprezentat de vârf
$ST[i]$ = numărul fiului stâng
$DR[i]$ = numărul fiului drept

```
procedure interopt(Q[1 .. n])
  {construiește arborele strategiei greedy de interclasare
   a șirurilor de lungimi Q[i] = q_i, 1 ≤ i ≤ n}
  H ← min-heap vid
  for i ← 1 to n do
     (Q[i], i) ⇒ H        {inserează în min-heap}
     LU[i] ← Q[i]; ST[i] ← 0; DR[i] ← 0
  for i ← n+1 to 2n–1 do
     (s, j)  ⇐ H          {extrage rădăcina lui H}
     (r, k)  ⇐ H          {extrage rădăcina lui H}
     ST[i] ← j; DR[i] ← k; LU[i] ← s+r
     (LU[i], i) ⇒ H       {inserează în min-heap}
```

În cazul cel mai nefavorabil, operațiile de inserare în min-heap și de extragere din min-heap necesită un timp în ordinul lui $\log n$ (revedeți Exercițiul 5.27). Restul operațiilor necesită un timp constant. Timpul total pentru *interopt* este deci în $O(n \log n)$.

6.4 Implementarea arborilor de interclasare

Transpunerea procedurii *interopt* într-un limbaj de programare prezintă o singură dificultate generată de utilizarea unui min-heap de perechi vârf-lungime. În limbajul C++, implementarea arborilor de interclasare este aproape o operație de rutină, deoarece clasa parametrică heap (Secțiunea 4.2.2) permite manipularea unor heap-uri cu elemente de orice tip în care este definit operatorul de comparare >. Altfel spus, nu avem decât să construim o clasă formată din perechi vârf-lungime (pondere) și să o completăm cu operatorul > corespunzător. Vom numi această clasă vp, adică vârf-pondere.

```
#ifndef __VP_H
#define __VP_H

#include <iostream.h>

class vp {
public:
  vp( int vf = 0, float pd = 0 ) { v = vf; p = pd; }

  operator int  ( ) const { return v; }
  operator float( ) const { return p; }

  int v; float p;
};

inline operator > ( const vp& a, const vp& b) {
  return a.p <  b.p;
}

inline istream& operator >>( istream& is, vp& element ) {
  is >> element.v >> element.p; element.v--;
  return is;
}

inline ostream& operator <<( ostream& os, vp& element ) {
  os << "{ " << (element.v+1) << "; " << element.p << " }";
  return os;
}
#endif
```

Scopul clasei `vp` (definită în fișierul `vp.h`) nu este de a introduce un nou tip de date, ci mai curând de a facilita manipularea structurii vârf-pondere, structură utilă și la reprezentarea grafurilor. Din acest motiv, nu există nici un fel de încapsulare, toți membrii fiind publici. Pentru o mai mare comoditate în utilizare, am inclus în definiție cei doi operatori de conversie, la `int`, respectiv la `float`, precum și operatorii de intrare/ieșire.

Nu ne mai rămâne decât să precizăm structura arborelui de interclasare. Cel mai simplu este să preluăm structura folosită în procedura *interopt* din Secțiunea 6.3: arborele este format din trei tablouri paralele, care conțin lungimea șirului reprezentat de vârful respectiv și indicii celor doi fii. Pentru o scriere mai compactă, vom folosi totuși o structură puțin diferită: un tablou de elemente de tip `nod`, fiecare `nod` conținând trei câmpuri corespunzătoare informațiilor de mai sus. Clasa `nod` este similară clasei `vp`, atât ca structură, cât și prin motivația introducerii ei.

```
class nod {
public:
  int lu; // lungimea
  int st; // fiul stang
  int dr; // fiul drept
};

inline ostream& operator <<( ostream& os, nod& nd ) {
  os << " <" << nd.st << "< "
     << nd.lu
     << " >" << nd.dr << "> ";
  return os;
}
```

În limbajul C++, funcția de construire a arborelui strategiei greedy se obține direct, prin transcrierea procedurii *interopt*.

```
tablou<nod> interopt( const tablou<int>& Q ) {
  int n = Q.size( );

  tablou<nod> A( 2 * n - 1 ); // arborele de interclasare
  heap  <vp>  H( 2 * n - 1 );

  for ( int i = 0; i < n; i++ ) {
    H.insert( vp(i, Q[i]) );
    A[i].lu = Q[i]; A[i].st = A[i].dr = -1;
  }
  for ( i = n; i < 2 * n - 1; i++ ) {
    vp s; H.delete_max( s );
    vp r; H.delete_max( r );
    A[i].st = s; A[i].dr = r;
    A[i].lu = (float)s + (float)r;
    H.insert( vp(i, A[i].lu) );
  }
  return A;
}
```

Funcția de mai sus conține două aspecte interesante:

- Constructorul `vp(int, float)` este invocat explicit în funcția de inserare în heap-ul `H`. Efectul acestei invocări constă în crearea unui obiect temporar de tip `vp`, obiect distrus după inserare. O notație foarte simplă ascunde deci și o anumită ineficiență, datorată creării și distrugerii obiectului temporar.
- Operatorul de conversie la `int` este invocat implicit în expresiile `A[i].st = s` și `A[i].dr = r`, iar în expresia `A[i].lu = (float)s + (float)r`, operatorul de conversie la `float` trebuie să fie specificat explicit. Semantica limbajului C++ este foarte clară relativ la conversii: cele utilizator au prioritate față de cele standard, iar ambiguitatea în selectarea conversiilor posibile este semnalată ca eroare. Dacă în primele două atribuiri conversia lui `s` și `r` la `int` este singura posibilitate, scrierea celei de-a treia sub forma

`A[i].lu = s + r` este ambiguă, expresia `s + r` putând fi evaluată atât ca `int` cât și ca `float`.

În final, nu ne mai rămâne decât să testăm funcția `interopt()`. Vom folosi un tablou `l` cu lungimi de șiruri, lungimi extrase din `stream`-ul standard de intrare.

```
main( ) {
  tablou<int> l;
  cout << "Siruri: "; cin >> l;
  cout << "Arborele de interclasare: ";
  cout << interopt( l ) << '\n';
  return 1;
}
```

Strategia de interclasare optimă pentru cele șase lungimi folosite ca exemplu în Secțiunea 6.3:

```
[ 6 ] 30 10 20 30 50 10
```

este:

```
Arborele de interclasare:  [11]:  <-1< 30 >-1>  <-1< 10 >-1>
<-1< 20 >-1>   <-1< 30 >-1>   <-1< 50 >-1>   <-1< 10 >-1>
<1< 20 >5>   <2< 40 >6>   <3< 60 >0>   <7< 90 >4>
<8< 150 >9>
```

Valoarea fiecărui nod este precedată de indicele fiului stâng și urmată de cel al fiului drept, indicele `-1` reprezentând legătura inexistentă. Formatele de citire și scriere ale tablourilor sunt cele stabilite în Secțiunea 4.1.3.

6.5 Coduri Huffman

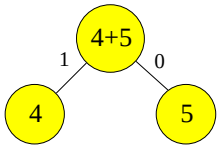
O altă aplicație a strategiei greedy și a arborilor binari cu lungime externă ponderată minimă este obținerea unei codificări cât mai compacte a unui text.

Un principiu general de codificare a unui șir de caractere este următorul: se măsoară frecvența de apariție a diferitelor caractere dintr-un eșantion de text și se atribuie cele mai scurte coduri, celor mai frecvente caractere, și cele mai lungi coduri, celor mai puțin frecvente caractere. Acest principiu stă, de exemplu, la baza codului Morse. Pentru situația în care codificarea este binară, există o metodă elegantă pentru a obține codul respectiv. Această metodă, descoperită de Huffman (1952) folosește o strategie greedy și se numește *codificarea Huffman*. O vom descrie pe baza unui exemplu.

Fie un text compus din următoarele litere (în paranteze figurează frecvențele lor de apariție):

S (10), I (29), P (4), O (9), T (5)

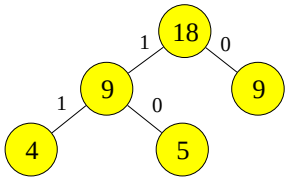
Conform metodei greedy, construim un arbore binar fuzionând cele două litere cu frecvențele cele mai mici. Valoarea fiecărui vârf este dată de frecvența pe care o reprezintă.



Etichetăm muchia stângă cu 1 și muchia dreaptă cu 0. Rearanjăm tabelul de frecvențe:

S (10), I (29), O (9), {P, T} (4+5 = 9)

Mulțimea {P, T} semnifică evenimentul reuniune a celor două evenimente independente corespunzătoare apariției literelor P și T. Continuăm procesul, obținând arborele



În final, ajungem la arborele din Figura 6.3, în care fiecare vârf terminal corespunde unei litere din text.

Pentru a obține codificarea binară a literei P, nu avem decât să scriem secvența de 0-uri și 1-uri în ordinea apariției lor pe drumul de la rădăcină către vârful corespunzător lui P: 1011. Procedăm similar și pentru restul literelor:

S (11), I (0), P (1011), O (100), T (1010)

Pentru un text format din n litere care apar cu frecvențele $f_1, f_2, \ldots, f_n$, un *arbore de codificare* este un arbore binar cu vârfurile terminale având valorile $f_1, f_2, \ldots, f_n$, prin care se obține o codificare binară a textului. Un arbore de codificare nu trebuie în mod necesar să fie construit după metoda greedy a lui Huffman, alegerea vârfurilor care sunt fuzionate la fiecare pas putându-se face

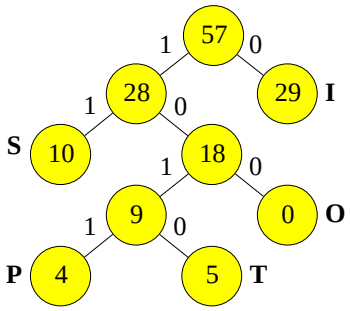


Figura 6.3 Arborele de codificare Huffman.

după diverse criterii. Lungimea externă ponderată a unui arbore de codificare este:

$$\sum_{i=1}^{n} a_i f_i$$

unde a_i este adîncimea vârfului terminal corespunzător literei i. Se observă că lungimea externă ponderată este egală cu numărul total de caractere din codificarea textului considerat. Codificarea cea mai compactă a unui text corespunde deci arborelui de codificare de lungime externă ponderată minimă. Se poate demonstra că arborele de codificare Huffman minimizează lungimea externă ponderată pentru toți arborii de codificare cu vârfurile terminale având valorile $f_1, f_2, \ldots, f_n$. Prin strategia greedy se obține deci întotdeauna codificarea binară cea mai compactă a unui text.

Arborii de codificare pe care i-am considerat în acestă secțiune corespund unei codificări de tip special: codificarea unei litere nu este prefixul codificării nici unei alte litere. O astfel de codificare este de tip *prefix*. Codul Morse nu face parte din această categorie. Codificarea cea mai compactă a unui șir de caractere poate fi întotdeauna obținută printr-un cod de tip prefix. Deci, concentrându-ne atenția asupra acestei categorii de coduri, nu am pierdut nimic din generalitate.

6.6 Arbori parțiali de cost minim

Fie $G = <V, M>$ un graf neorientat conex, unde V este mulțimea vârfurilor și M este mulțimea muchiilor. Fiecare muchie are un *cost* nenegativ (sau o *lungime* nenegativă). Problema este să găsim o submulțime $A \subseteq M$, astfel încât toate vârfurile din V să rămînă conectate atunci când sunt folosite doar muchii din A,

iar suma lungimilor muchiilor din A să fie cât mai mică. Căutăm deci o submulțime A de cost total minim. Această problemă se mai numește și *problema conectării orașelor cu cost minim*, având numeroase aplicații.

Graful parțial $<V, A>$ este un arbore (Exercițiul 6.11) și este *numit arborele parțial de cost minim* al grafului G (*minimal spanning tree*). Un graf poate avea mai mulți arbori parțiali de cost minim și acest lucru se poate verifica pe un exemplu.

Vom prezenta doi algoritmi greedy care determină arborele parțial de cost minim al unui graf. În terminologia metodei greedy, vom spune că o mulțime de muchii este o *soluție*, dacă constituie un arbore parțial al grafului G, și este *fezabilă*, dacă nu conține cicluri. O mulțime fezabilă de muchii este *promițătoare*, dacă poate fi completată pentru a forma soluția optimă. O muchie *atinge* o mulțime dată de vârfuri, dacă exact un capăt al muchiei este în mulțime. Următoarea proprietate va fi folosită pentru a demonstra corectitudinea celor doi algoritmi.

Proprietatea 6.2 Fie $G = <V, M>$ un graf neorientat conex în care fiecare muchie are un cost nenegativ. Fie $W \subset V$ o submulțime strictă a vârfurilor lui G și fie $A \subseteq M$ o mulțime promițătoare 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 promițătoare.

Demonstrație: Fie B un arbore parțial de cost minim al lui G, astfel încât $A \subseteq B$ (adică, muchiile din A sunt conținute în arborele B). Un astfel de B trebuie să existe, deoarece A este promițătoare. Dacă $m \in B$, nu mai rămâne nimic de demonstrat. Presupunem că $m \notin B$. Adăugându-l pe m la B, obținem exact un ciclu (Exercițiul 3.2). În acest ciclu, deoarece m atinge W, trebuie să mai existe cel puțin o muchie m' care atinge și ea pe W (altfel, ciclul nu se închide). Eliminându-l pe m', ciclul dispare și obținem un nou arbore parțial 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 și el un arbore parțial de cost minim al lui G, care include pe m. Observăm că $A \subseteq B'$ deoarece muchia m', care atinge W, nu poate fi în A. Deci, $A \cup \{m\}$ este promițătoare. ■

Mulțimea inițială a candidaților este M. Cei doi algoritmi greedy aleg muchiile una câte una într-o anumită ordine, această ordine fiind specifică fiecărui algoritm.

6.6.1 Algoritmul lui Kruskal

Arborele parțial de cost minim poate fi construit muchie, cu muchie, după următoarea metodă a lui Kruskal (1956): se alege întâi muchia de cost minim, iar

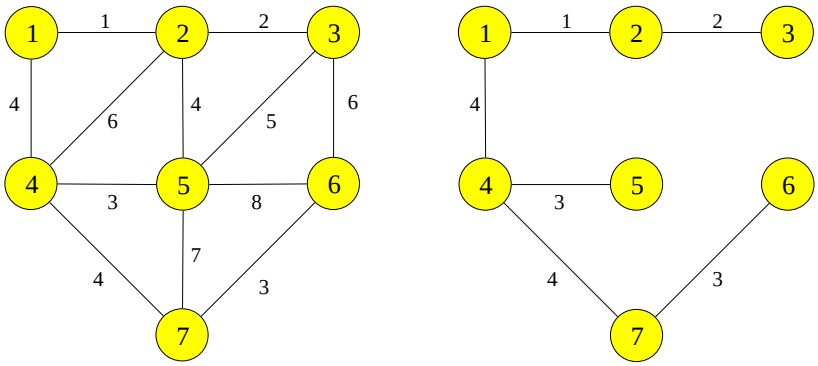


Figura 6.4 Un graf și arborele său parțial de cost minim.

apoi se adaugă repetat muchia de cost minim nealeasă anterior și care nu formează cu precedentele un ciclu. Alegem astfel #*V*–1 muchii. Este ușor de dedus că obținem în final un arbore (revedeți Exercițiul 3.2). Este însă acesta chiar arborele parțial de cost minim căutat?

Înainte de a răspunde la întrebare, să considerăm, de exemplu, graful din Figura 6.4a. Ordonăm crescător (în funcție 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} și apoi aplicăm algoritmul. Structura componentelor conexe este ilustrată, pentru fiecare pas, în Tabelul 6.1.

Mulțimea *A* este inițial vidă și se completează pe parcurs cu muchii acceptate

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

Tabelul 6.1 Algoritmul lui Kruskal aplicat grafului din Figura 6.4a.

(care nu formează un ciclu cu muchiile deja existente în A). În final, mulțimea A va conține muchiile {1, 2}, {2, 3}, {4, 5}, {6, 7}, {1, 4}, {4, 7}. La fiecare pas, graful parțial $<V, A>$ formează o pădure de componente conexe, obținută din pădurea precedentă unind două componente. Fiecare componentă conexă este la rândul ei un arbore parțial de cost minim pentru vârfurile pe care le conectează. Inițial, fiecare vârf formează o componentă conexă. La sfârșit, vom avea o singură componentă conexă, care este arborele parțial de cost minim căutat (Figura 6.4b).

Ceea ce am observat în acest caz particular este valabil și pentru cazul general, din Proprietatea 6.2 rezultând:

Proprietatea 6.3 În algoritmul lui Kruskal, la fiecare pas, graful parțial $<V, A>$ formează o pădure de componente conexe, în care fiecare componentă conexă este la rândul ei un arbore parțial de cost minim pentru vârfurile pe care le conectează. În final, se obține arborele parțial de cost minim al grafului G. ■

Pentru a implementa algoritmul, trebuie să putem manipula submulțimile formate din vârfurile componentelor conexe. Folosim pentru aceasta o structură de mulțimi disjuncte și procedurile de tip *find* și *merge* (Secțiunea 3.5). În acest caz, este preferabil să reprezentăm graful ca o listă de muchii cu costul asociat lor, astfel încât să putem ordona această listă în funcție de cost. Iată algoritmul:

```
function Kruskal(G = <V, M>)
    {inițializare}
    sortează M crescător în funcție de cost
    n ← #V
    A ← ∅ {va conține muchiile arborelui parțial de cost minim}
    inițializează n mulțimi disjuncte conținând
        fiecare câte un element din V

    {buclă greedy}
    repeat
        {u, v} ← muchia de cost minim care
                    încă nu a fost considerată
        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 și m muchii, presupunând că se folosesc procedurile *find*3 și *merge*3, numărul de operații pentru cazul cel mai nefavorabil este în:

Pasul	**Muchia considerată**	***U***
inițializare	—	{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 6.2 Algoritmul lui Prim aplicat grafului din Figura 6.4a.

- $O(m \log m)$ pentru a sorta muchiile. Deoarece $m \leq n(n-1)/2$, rezultă $O(m \log m) \subseteq O(m \log n)$. Mai mult, graful fiind conex, din $n-1 \leq m$ rezultă și $O(m \log n) \subseteq O(m \log m)$, deci $O(m \log m) = O(m \log n)$.
- $O(n)$ pentru a inițializa cele n mulțimi disjuncte.
- Cele cel mult $2m$ operații *find*3 și $n-1$ operații *merge*3 necesită un timp în $O((2m+n-1) \lg^* n)$, după cum am specificat în Capitolul 3. Deoarece $O(\lg^* n) \subseteq O(\log n)$ și $n-1 \leq m$, acest timp este și în $O(m \log n)$.
- $O(m)$ pentru restul operațiilor.

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

O alta variantă este să păstrăm muchiile într-un min-heap. Obținem astfel un nou algoritm, în care inițializarea se face într-un timp în $O(m)$, iar fiecare din cele $n-1$ extrageri ale unei muchii minime se face într-un timp în $O(\log m) = O(\log n)$. Pentru cazul cel mai nefavorabil, ordinul timpului rămâne același cu cel al vechiului algoritm. Avantajul folosirii min-heap-ului apare atunci când arborele parțial de cost minim este găsit destul de repede și un număr considerabil de muchii rămân netestate. În astfel de situații, algoritmul vechi pierde timp, sortând în mod inutil și aceste muchii.

6.6.2 Algoritmul lui Prim

Cel de-al doilea algoritm greedy pentru determinarea arborelui parțial de cost minim al unui graf se datorează lui Prim (1957). În acest algoritm, la fiecare pas, mulțimea A de muchii alese împreună cu mulțimea U a vârfurilor pe care le conectează formează un arbore parțial de cost minim pentru subgraful $<U, A>$ al lui G. Inițial, mulțimea U a vârfurilor acestui arbore conține un singur vârf oarecare din V, care va fi rădăcina, iar mulțimea A a muchiilor este vidă. La fiecare pas, se alege o muchie de cost minim, care se adaugă la arborele precedent, dând naștere unui nou arbore parțial de cost minim (deci, exact una

dintre extremitățile acestei muchii este un vârf în arborele precedent). Arborele parțial de cost minim crește "natural", cu câte o ramură, pînă când va atinge toate vârfurile din V, adică pînă când $U = V$. Funcționarea algoritmului, pentru exemplul din Figura 6.4a, este ilustrată în Tabelul 6.2. La sfârșit, A va conține aceleași muchii ca și în cazul algoritmului lui Kruskal. Faptul că algoritmul funcționează întotdeauna corect este exprimat de următoarea proprietate, pe care o puteți demonstra folosind Proprietatea 6.2.

Proprietatea 6.4 În algoritmul lui Prim, la fiecare pas, $<U, A>$ formează un arbore parțial de cost minim pentru subgraful $<U, A>$ al lui G. În final, se obține arborele parțial de cost minim al grafului G. ■

Descrierea formală a algoritmului este dată în continuare.

function *Prim-formal*($G = <V, M>$)
  {inițializare}
  $A \leftarrow \emptyset$ {va conține muchiile arborelui parțial de cost minim}
  $U \leftarrow$ {un vârf oarecare din V}
  {buclă greedy}
  while $U \neq V$ **do**
    găsește $\{u, v\}$ de cost minim astfel ca $u \in V \setminus U$ și $v \in U$
    $A \leftarrow A \cup \{\{u, v\}\}$
    $U \leftarrow U \cup \{u\}$
  return A

Pentru a obține o implementare simplă, presupunem că: vârfurile din V sunt numerotate de la 1 la n, $V = \{1, 2, \ldots, n\}$; matricea simetrică C dă costul fiecărei muchii, cu $C[i, j] = +\infty$, dacă muchia $\{i, j\}$ nu există. Folosim două tablouri paralele. Pentru fiecare $i \in V \setminus U$, *vecin*[i] conține vârful din U, care este conectat de i printr-o muchie de cost minim; *mincost*[i] dă acest cost. Pentru $i \in U$, punem *mincost*[i] = –1. Mulțimea U, în mod arbitrar inițializată cu {1}, nu este reprezentată explicit. Elementele *vecin*[1] și *mincost*[1] nu se folosesc.

```
function Prim(C[1 .. n, 1 .. n])
   {inițializare; numai vârful 1 este în U}
   A ← ∅
   for i ← 2 to n do  vecin[i]    ← 1
                      mincost[i] ← C[i, 1]
   {buclă 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 principală se execută de n–1 ori și, la fiecare iterație, buclele **for** din interior necesită un timp în $O(n)$. Algoritmul *Prim* necesită, deci, un timp în $O(n^2)$. Am văzut că timpul pentru algoritmul lui Kruskal este în $O(m \log n)$, unde $m = \#M$. Pentru un graf *dens* (adică, cu foarte multe muchii), se deduce că m se apropie de $n(n–1)/2$. În acest caz, algoritmul *Kruskal* necesită un timp în $O(n^2 \log n)$ și algoritmul *Prim* este probabil mai bun. Pentru un graf *rar* (adică, cu un număr foarte mic de muchii), m se apropie de n și algoritmul *Kruskal* necesită un timp în $O(n \log n)$, fiind probabil mai eficient decât algoritmul *Prim*.

6.7 Implementarea algoritmului lui Kruskal

Funcția care implementează algoritmul lui Kruskal în limbajul C++ este aproape identică cu procedura *Kruskal* din Secțiunea 6.6.1.

```
tablou<muchie> Kruskal( int n, const tablou<muchie>& M ) {
  heap<muchie> h( M );

  tablou<muchie> A( n - 1 ); int nA = 0;
  set            s( n );

  do {
    muchie m;
    if ( !h.delete_max( m ) )
      { cerr << "\n\nKruskal -- heap vid.\n\n"; return A = 0; }
```

```
      int ucomp = s.find3( m.u ),
          vcomp = s.find3( m.v );

      if ( ucomp != vcomp ) {
        s.merge3( ucomp, vcomp );
        A[ nA++ ] = m;
      }
    } while ( nA != n - 1 );

    return A;
  }
```

Diferențele care apar sunt mai curând precizări suplimentare, absolut necesare în trecerea de la descrierea unui algoritm la implementarea lui. Astfel, graful este transmis ca parametru, prin precizarea numărului de vârfuri și a muchiilor. Pentru muchii, reprezentate prin cele două vârfuri și costul asociat, am preferat în locul listei, structura simplă de tablou `M`, structură folosită și la returnarea arborelui de cost minim `A`.

Operația principală efectuată asupra muchiilor este alegerea muchiei de cost minim care încă nu a fost considerată. Pentru implementarea acestei operații, folosim un min-heap. La fiecare iterație, se extrage din heap muchia de cost minim și se încearcă inserarea ei în arborele `A`.

Rulând programul

```
main( ) {
  int n;
  cout << "\nVarfuri... ";
  cin >> n;

  tablou<muchie> M;
  cout << "\nMuchiile si costurile lor... ";
  cin >> M;

  cout << "\nArborele de cost minim Kruskal:\n";
  cout <<  Kruskal( n, M ) << '\n';
  return 1;
}
```

pentru graful din Figura 6.4a, obținem următoarele rezultate:

```
Arborele de cost minim Kruskal:
  [6]: { 1, 2; 1 } { 2, 3; 2 } { 4, 5; 3 } { 6, 7; 3 }
  { 1, 4; 4 } { 4, 7; 4 }
```

Clasa `muchie`, folosită în implementarea algoritmului lui Kruskal, trebuie să permită:

- Inițializarea obiectelor, inclusiv cu valori implicite (inițializare utilă la construirea tablourilor de muchii).
- Compararea obiectelor în funcție de cost (operație folosită de min-heap).
- Operații de citire și scriere (invocate indirect de operatorii respectivi din clasa `tablou<T>`).

Pornind de la aceste cerințe, se obține următoarea implementare, conținută în fișierul `muchie.h`.

```
#ifndef __MUCHIE_H
#define __MUCHIE_H

class muchie {
public:
  muchie( int iu = 0, int iv = 0, float ic = 0. )
    { u = iu; v = iv; cost = ic; }

  int u, v;
  float cost;
};

inline operator >( const muchie& a, const muchie& b ) {
  return a.cost < b.cost;
}

inline istream& operator >>( istream& is, muchie& m ) {
  is >> m.u >> m.v >> m.cost; m.u--; m.v--;
  return is;
}

inline ostream& operator<< ( ostream& os, muchie& m ) {
  return os << "{ " << (m.u+1) << ", " << (m.v+1)
                              << "; " << m.cost << " }";
}

#endif
```

În ceea ce privește clasa `set`, folosită și ea în implementarea algoritmului *Kruskal*, vom urma precizările din Secțiunea 3.5 relative la manipularea mulțimilor disjuncte. Încapsularea, într-o clasă, a structurii de mulțimi disjuncte și a procedurilor *find*3 și *merge*3 nu prezintă nici un fel de dificultăți. Vom prezenta, totuși, implementarea clasei `set`, deoarece spațiul de memorie folosit este redus la jumătate.

La o analiză mai atentă a procedurii *merge*3, observăm că tabloul înălțimii arborilor este folosit doar pentru elementele care sunt și etichete de mulțimi (vezi Exercițiul 3.13). Aceste elemente, numite *elemente canonice*, sunt rădăcini ale arborilor respectivi. Altfel spus, un element canonic nu are tată și valoarea lui este folosită doar pentru a-l diferenția de elementele care nu sunt canonice. În

Secțiunea 3.5, elementele canonice sunt diferențiate prin faptul că `set[i]` are valoarea `i`. Având în vedere că `set[i]` este indicele în tabloul `set` al tatălui elementului `i`, putem asocia elementelor canonice proprietatea `set[i] < 0`. Prin această convenție, valoarea absolută a elementelor canonice poate fi oarecare. Atunci, de ce să nu fie chiar înălțimea arborelui?

În concluzie, pentru reprezentarea structurii de mulțimi disjuncte, este necesar un singur tablou, numit `set`, cu tot atâtea elemente câte are și mulțimea. Valorile inițiale ale elemetelor tabloului `set` sunt `-1`. Aceste inițializări vor fi realizate prin constructor. Interfața publică a clasei `set` trebuie să conțină funcțiile `merge3()` și `find3()`, adaptate corepunzător. Tratarea situațiilor de excepție care pot să apară la invocarea acestor funcții (indici de mulțimi în afara intervalului permis) se realizează prin activarea procedurii de verificare a indicilor în tabloul `set`.

Aceste considerente au condus la următoarele definiții ale funcțiilor membre din clasa `set`.

```
#include "set.h"

set::set( int n ): set( n ) {
  set.vOn( );
  for ( int i = 0; i < n; i++ )
    set[ i ] = -1;
}

void set::merge3( int a, int b ) {
  // sunt a si b etichete de multimi?
  if ( set[ a ] >= 0 ) a = find3( a );
  if ( set[ b ] >= 0 ) b = find3( b );

  // sunt multimile a si b diferite?
  if ( a == b ) return;

  // reuniunea propriu-zisa
  if ( set[ a ] == set[ b ] ) set[ set[ b ] = a ]--;
  else if ( set[ a ] < set[ b ] )  set[ b ] = a;
       else                        set[ a ] = b;

  return;
}
```

```
int set::find3( int x ) {
  int r = x;
  while ( set[ r ] >= 0 )
    r = set[ r ];

  int i = x;
  while ( i != r )
    { int j = set[ i ]; set[ i ] = r; i = j; }

  return r;
}
```

Fişierul header `set.h` este:

```
#ifndef __SET_H
#define __SET_H

#include "heap.h"

class set {
public:
  set( int );
  void merge3( int, int );
  int  find3 ( int );

private:
  tablou<int> set;
};

#endif
```

6.8 Cele mai scurte drumuri care pleacă din același punct

Fie $G = <V, M>$ un graf orientat, unde V este mulțimea vârfurilor și M este mulțimea muchiilor. Fiecare muchie are o lungime nenegativă. Unul din vârfuri este desemnat ca vârf *sursă*. Problema este să determinăm lungimea celui mai scurt drum de la sursă către fiecare vârf din graf.

Vom folosi un algoritm greedy, datorat lui Dijkstra (1959). Notăm cu C mulțimea vârfurilor disponibile (candidații) și cu S mulțimea vârfurilor deja selectate. În fiecare moment, S conține acele vârfuri a căror distanță minimă de la sursă este deja cunoscută, în timp ce mulțimea C conține toate celelalte vârfuri. La început, S conține doar vârful sursă, iar în final S conține toate vârfurile grafului. La fiecare pas, adăugăm în S acel vârf din C a cărui distanță de la sursă este cea mai mică.

Spunem că un drum de la sursă către un alt vârf este *special*, dacă toate vârfurile intermediare de-a lungul drumului aparțin lui S. Algoritmul lui Dijkstra lucrează în felul următor. La fiecare pas al algoritmului, un tablou D conține lungimea celui mai scurt drum special către fiecare vârf al grafului. După ce adăugăm un nou vârf v la S, cel mai scurt drum special către v va fi, de asemenea, cel mai scurt dintre toate drumurile către v. Când algoritmul se termină, toate vârfurile din graf sunt în S, deci toate drumurile de la sursă către celelalte vârfuri sunt speciale și valorile din D reprezintă soluția problemei.

Presupunem, pentru simplificare, că vârfurile sunt numerotate, $V = \{1, 2, \ldots, n\}$, vârful 1 fiind sursa, și că matricea L dă lungimea fiecărei muchii, cu $L[i, j] = +\infty$, dacă muchia (i, j) nu există. Soluția se va construi în tabloul $D[2 .. n]$. Algoritmul este:

```
function Dijkstra(L[1 .. n, 1 .. n])
    {inițializare}
    C ← {2, 3, ..., n}        {S = V \C există doar implicit}
    for i ← 2 to n do D[i] ← L[1, i]
    {bucla greedy}
    repeat n–2 times
        v ← vârful din C care minimizează D[v]
        C ← C \ {v}      {și, implicit, S ← S ∪ {v}}
        for fiecare w ∈ C do
            D[w] ← min(D[w], D[v]+L[v, w])
    return D
```

Pentru graful din Figura 6.5, pașii algoritmului sunt prezentați în Tabelul 6.3.

Observăm că D nu se schimbă dacă mai efectuăm o iterație pentru a-l scoate și pe $\{2\}$ din C. De aceea, bucla greedy se repetă de doar $n–2$ ori.

Se poate demonstra următoarea proprietate:

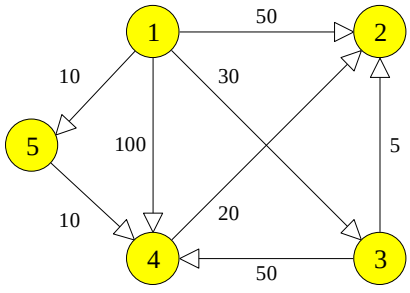


Figura 6.5 Un graf orientat.

Pasul	v	C	D
inițializare	—	{2, 3, 4, 5}	[50, 30, 100, 10]
1	5	{2, 3, 4}	[50, 30, 20, 10]
2	4	{2, 3}	[40, 30, 20, 10]
3	3	{2}	[35, 30, 20, 10]

Tabelul 6.3 Algoritmul lui Dijkstra aplicat grafului din Figura 6.5.

Proprietatea 6.5. În algoritmul lui Dijkstra, dacă un vârf i

i) este în S, atunci $D[i]$ dă lungimea celui mai scurt drum de la sursă către i;

ii) nu este în S, atunci $D[i]$ dă lungimea celui mai scurt drum special de la sursă către i. ■

La terminarea algoritmului, toate vârfurile grafului, cu excepția unuia, sunt în S. Din proprietatea precedentă, rezultă că algoritmul lui Dijkstra funcționează corect.

Dacă dorim să aflăm nu numai lungimea celor mai scurte drumuri, dar și pe unde trec ele, este suficient să adăugăm un tablou $P[2 .. n]$, unde $P[v]$ conține numărul nodului care îl precede pe v în cel mai scurt drum. Pentru a găsi drumul complet, nu avem decât să urmărim, în tabloul P, vârfurile prin care trece acest drum, de la destinație la sursă. Modificările în algoritm sunt simple:

- inițializează $P[i]$ cu 1, pentru $2 \le i \le n$
- conținutul buclei **for** cea mai interioară se înlocuiește cu

  if $D[w] > D[v]+L[v, w]$ **then** $D[w] \leftarrow D[v]+L[v, w]$
  $P[w] \leftarrow v$

- bucla **repeat** se execută de $n-1$ ori

Să presupunem că aplicăm algoritmul *Dijkstra* asupra unui graf cu n vârfuri și m muchii. Inițializarea necesită un timp în $O(n)$. Alegerea lui v din bucla **repeat** presupune parcurgerea tuturor vârfurilor conținute în C la iterația respectivă, deci a $n-1, n-2, \ldots, 2$ vârfuri, ceea ce necesită în total un timp în $O(n^2)$. Bucla **for** interioară efectuează $n-2, n-3, \ldots, 1$ iterații, totalul fiind tot în $O(n^2)$. Rezultă că algoritmul Dijkstra necesită un timp în $O(n^2)$.

Încercăm să îmbunătățim acest algoritm. Vom reprezenta graful nu sub forma matricii de adiacență L, ci sub forma a n liste de adiacență, conținând pentru fiecare vârf lungimea muchiilor care pleacă din el. Bucla **for** interioară devine astfel mai rapidă, deoarece putem să considerăm doar vârfurile w adiacente lui v. Aceasta nu poate duce la modificarea ordinului timpului total al algoritmului,

dacă nu reușim să scădem și ordinul timpului necesar pentru alegerea lui *v* din bucla **repeat**. De aceea, vom ține vârfurile *v* din *C* într-un min-heap, în care fiecare element este de forma (*v*, *D*[*v*]), proprietatea de min-heap referindu-se la valoarea lui *D*[*v*]. Numim algoritmul astfel obținut *Dijkstra-modificat*. Să îl analizăm în cele ce urmează.

Inițializarea min-heap-ului necesită un timp în $O(n)$. Instrucțiunea "$C \leftarrow C \setminus \{v\}$" constă în extragerea rădăcinii min-heap-ului și necesită un timp în $O(\log n)$. Pentru cele $n-2$ extrageri este nevoie de un timp în $O(n \log n)$.

Pentru a testa dacă "$D[w] > D[v]+L[v, w]$", bucla **for** interioară constă acum în inspectarea fiecărui vârf *w* din *C* adiacent lui *v*. Fiecare vârf *v* din *C* este introdus în *S* exact o dată și cu acest prilej sunt testate exact muchiile adiacente lui; rezultă că numărul total de astfel de testări este de cel mult *m*. Dacă testul este adevărat, trebuie să îl modificăm pe *D*[*w*] și să operăm un *percolate* cu *w* în min-heap, ceea ce necesită din nou un timp în $O(\log n)$. Timpul total pentru operațiile *percolate* este deci în $O(m \log n)$.

În concluzie, algoritmul *Dijkstra-modificat* necesită un timp în $O(\max(n, m) \log n)$. Dacă graful este conex, atunci $m \geq n$ și timpul este în $O(m \log n)$. Pentru un graf rar este preferabil să folosim algoritmul *Dijkstra-modificat*, iar pentru un graf dens algoritmul *Dijkstra* este mai eficient.

Este ușor de observat că, într-un graf *G* neorientat conex, muchiile celor mai scurte drumuri de la un vârf *i* la celelalte vârfuri formează un *arbore parțial al celor mai scurte drumuri* pentru *G*. Desigur, acest arbore depinde de alegerea rădăcinii *i* și el diferă, în general, de arborele parțial de cost minim al lui *G*.

Problema găsirii celor mai scurte drumuri care pleacă din același punct se poate pune și în cazul unui graf neorientat.

6.9 Implementarea algoritmului lui Dijkstra

Această secțiune este dedicată implementării algoritmului *Dijkstra-modificat* pentru determinarea celor mai scurte drumuri care pleacă din același vârf. După cum am văzut, acest algoritm este de preferat în cazul grafurilor rare, timpul lui fiind în ordinul lui $O(m \log n)$, unde *m* este numărul de muchii, iar *n* numărul de vârfuri ale unui graf conex.

În implementarea noastră, tipul de date "fundamental" este clasa vp (vârf-pondere), definită cu ocazia implementării arborilor de interclasare. Vom folosi această clasă pentru:

- Min-heap-ul C format din perechi (v, d), ponderea d fiind lungimea celui mai scurt drum special de la vârful sursă la vârful v.

- Reprezentarea grafului `G` prin liste de adiacență. Pentru fiecare vârf `v`, perechea (`w`, `l`) este muchia de lungime `l` cu extremitățile în `v` și `w`.
- Tabloul `P`, al rezultatelor. Elementul `P[i]`, de valoare (`v`, `d`), reprezintă vârful `v` care precede vârful `i` în cel mai scurt drum de la vârful sursă, `d` fiind lungimea acestui drum.

Graful `G` este implementat ca un tablou de liste de elemente de tip vârf-pondere. Tipul `graf`, introdus prin

```
typedef tablou< lista<vp> > graf;
```

este un sinonim pentru această structură.

Definiția de mai sus merită o clipă de atenție, deoarece exemplifică una din puținele excepții lexicale din C++. În limbajul C++, ca și în limbajul C, noțiunea de separator este inexistentă. Separarea atomilor lexicali ai limbajului (identificatori, operatori, cuvinte cheie, constante) prin caracterele "albe" spațiu sau tab este opțională. Totuși, în `typedef`-ul anterior, cele două semne `>` trebuie separate, pentru a nu fi interpretate ca operatorul de decalare `>>`.

Manipularea grafului `G`, definit ca `graf G`, implică fixarea unui vârf și apoi operarea asupra listei asociate vârfului respectiv. Pentru o simplă parcurgere, nu avem decât să definim iteratorul `iterator<vp> g` și să-l inițializăm cu una din listele de adiacență, de exemplu cu cea corespunzătoare vârfului 2: `g = G[ 2 ];`.

Dacă `w` este un obiect de tip `vp`, atunci, prin instrucțiunea

```
while( g( w ) ) {
  // ...
}
```

obiectul `w` va conține, rând pe rând, toate extremitățile și lungimile muchiilor care pleacă din vârful 2.

Structura obiectului `graf G` asociat grafului din Figura 6.5, structură tipărită prin `cout << G`, este:

```
[5]:  { { 5; 10 } { 4; 100 } { 3; 30 } { 2; 50 } }
      { }
      { { 4; 50 } { 2; 5 } }
      { { 2; 20 } }
      { { 4; 10 } }
```

Executarea acestei instrucțiuni implică invocarea operatorilor de inserare `<<` ai tuturor celor 3 clase implicate, adică `vp`, `tablou<T>` și `lista<E>`.

Citirea grafului `G` se realizează prin citirea muchiilor și inserarea lor în listele de adiacență. În acest scop, vom folosi aceeași clasă `muchie`, utilizată și în implementarea algoritmului lui Kruskal:

```
int n, m = 0;  // #varfuri si #muchii
muchie M;

cout << "Numarul de varfuri... "; cin >> n;
graf G( n );

cout << "Muchiile... ";
while( cin >> M ) {
  // aici se poate verifica corectitudinea muchiei M
  G[ M.u ].insert( vp( M.v, M.cost ) );
  m++;
}
```

Algoritmul *Dijkstra-modificat* este implementat prin funcția

```
tablou<vp> Dijkstra( const graf& G, int m, int s );
```

funcție care returnează tabloul `tablou<vp> P(n)`. În lista de argumente a acestei funcții, `m` este numărul de muchii, iar `s` este vârful sursă. După cum am menționat, `P[i].v` (sau `(int)P[i]`) este vârful care precede vârful `i` pe cel mai scurt drum de la sursă către `i`, iar `P[i].p` (sau `(float)P[i]`) este lungimea acestui drum. De exemplu, pentru același graf din Figura 6.5, secvența:

```
for ( int s = 0; s < n; s++ ) {
    cout << "\nCele mai scurte drumuri de la varful "
         << (s + 1) << " sunt:\n";
    cout << Dijkstra( G, m, s ) << '\n';
  }
```

generează rezultatele:

```
Cele mai scurte drumuri de la varful 1 sunt:
[5]: { 1; 0 } { 3; 35 } { 1; 30 } { 5; 20 }
     { 1; 10 }

Cele mai scurte drumuri de la varful 2 sunt:
[5]: { 2; 3.37e+38 } { 1; 0 } { 2; 3.37e+38 }
     { 2; 3.37e+38 } { 2; 3.37e+38 }

Cele mai scurte drumuri de la varful 3 sunt:
[5]: { 3; 3.37e+38 } { 3; 5 } { 1; 0 } { 3; 50 }
     { 3; 3.37e+38 }
```

```
Cele mai scurte drumuri de la varful 4 sunt:
[5]: { 4; 3.37e+38 } { 4; 20 } { 4; 3.37e+38 }
     { 1; 0 } { 4; 3.37e+38 }

Cele mai scurte drumuri de la varful 5 sunt:
[5]: { 5; 3.37e+38 } { 4; 30 } { 5; 3.37e+38 }
     { 5; 10 } { 1; 0 }
```

unde 3.37e+38 este constanta `MAXFLOAT` din fișierul header `<values.h>`. `MAXFLOAT` este o aproximare rezonabilă pentru $+\infty$, fiind cel mai mare număr real admis de calculatorul pentru care se compilează programul.

Datele locale funcției `Dijkstra()` sunt heap-ul `heap<vp> C(n + m)` și tabloul `tablou<vp> P(n)` al celor mai scurte drumuri (incluzând și distanțele respective) de la fiecare din cele `n` vârfuri la vârful sursă. Inițial, distanțele din `P[s]` sunt $+\infty$ (constanta `MAXFLOAT` din `<values.h>`), exceptând vârful `s` și celelalte vârfuri adiacente lui `s`, vârfuri incluse și în `heap`-ul `C`. Inițializarea variabilelor `P` și `C` este realizată prin secvența:

```
vp w;

// initializare
for ( int i = 0; i < n; i++ )
  P[ i ] = vp( s, MAXFLOAT );
for ( iterator<vp> g = G[ s ]; g( w ); )
  { C.insert( w ); P[ w ] = vp( s, w ); }
P[ s ] = vp( 0, 0 );
```

Se observă aici invocarea explicită a constructorului clasei `vp` pentru inițializarea elementelor tabloului `P`. Din păcate, inițializarea nu este directă, ci prin intermediul unui obiect temporar de tip `vp`, obiect distrus după atribuire. Inițializarea directă este posibilă, dacă vom completa clasa `vp` cu o funcție de genul

```
vp& vp::set( int varf, float pondere )
  { v = varf; p = pondere; return *this; }
```

sau cu un operator

```
vp& vp::operator ( )( int varf, float pondere )
  { v = varf; p = pondere; return *this; }
```

Deși era mai natural să folosim operatorul de atribuire `=`, nu l-am putut folosi deoarece este operator binar, iar aici avem nevoie de 3 operanzi: în membrul stâng obiectul invocator și în membrul drept vârful, împreună cu ponderea. Folosind noul operator `()`, secvența de inițializare devine mai scurtă și mai eficientă:

```
vp w;

// initializare
for ( int i = 0; i < n; i++ )
  P[ i ]( s, MAXFLOAT );
for ( iterator<vp> g = G[ s ]; g( w ); )
  { C.insert( w ); P[ w ]( s, w ); }
P[ s ]( 0, 0 );
```

Bucla greedy a funcției `Dijkstra()`

```
vp v;
float dw;

// bucla greedy
for ( i = 1; i < n - 1; i++ ) {
  C.delete_max( v ); g = G[ v ];
  while ( g( w ) )
    if ( (float)P[ w ] > (dw = (float)P[ v ] + (float)w) )
      C.insert( vp( w, P[ w ]( v, dw ) ) );
}
```

se obține prin traducerea directă a descrierii algoritmului *Dijkstra-modificat*. Fiind dificil să căutăm în heap-ul `C` elemente (`w`, `D[w]`) după valoarea lui `w`, am inlocuit următoarele operații:

i) căutarea elementului (`w`, `D[w]`) pentru un `w` fixat

ii) modificarea valorii `D[w]`

iii) refacerea proprietății de heap

cu o simplă inserare în heap a unui nou element (`w`, `D[w]`), `D[w]` fiind modificat corespunzător. Din păcate, această simplificare poate mări heap-ul, deoarece există posibilitatea ca pentru fiecare muchie să fie inserat câte un nou element. Numărul de elemente din heap va fi însă totdeauna mai mic decât $n + m$. Timpul algoritmului rămâne în $O(m \log n)$.

Crearea unui obiect temporar la inserarea în heap este justificată aici chiar prin algoritm. Conform precizărilor de mai sus, actualizarea distanțelor se realizează indirect, prin inserarea unui nou obiect. Să remarcăm și înlocuirea tabloului redundant `D` cu membrul `float` din tabloul `P`.

În final, după executarea de `n-2` ori a buclei greedy, funcția `Dijkstra()` trebuie să returneze tabloul `P`:

```
return P;
```

Dacă secvențele prezentate până acum nu vă sunt suficiente pentru a scrie funcția `Dijkstra()` și programul de test, iată forma lor completă:

```
#include <iostream.h>
#include <values.h>

#include "tablou.h"
#include "heap.h"
#include "muchie.h"
#include "lista.h"
#include "vp.h"

typedef tablou< lista<vp> > graf;

tablou<vp> Dijkstra( const graf& G, int m, int s ) {
  int n = G.size( );  // numarul de varfuri ale grafului G

  heap<vp> C( m );
  tablou<vp> P( n );

  vp v, w;   // muchii
  float dw;  // distanta

  // initializare
  for ( int i = 0; i < n; i++ )
    P[ i ]( s, MAXFLOAT );
  for ( iterator<vp> g = G[ s ]; g( w ); )
    C.insert( w ); P[ w ]( s, w );
  P[ s ]( 0, 0 );

  // bucla greedy
  for ( i = 1; i < n - 1; i++ ) {
    C.delete_max( v ); g = G[ v ];
    while ( g( w ) )
      if ( (float)P[ w ] > ( dw = (float)P[ v ] + (float)w ) )
        C.insert( vp( w, P[ w ]( v, dw ) ) );
  }
  return P;
}

main( ) {
  int n, m = 0;  // #varfuri si #muchii
  muchie M;

  cout << "Numarul de varfuri... "; cin >> n;
  graf G( n );
```

```
    cout << "Muchiile... ";
    while( cin >> M ) {
      // aici se poate verifica corectitudinea muchiei M
      G[ M.u ].insert( vp( M.v, M.cost ) );
      m++;
    }

    cout << "\nListele de adiacenta:\n"; cout << G << '\n';

    for ( int s = 0; s < n; s++ ) {
      cout << "\nCele mai scurte drumuri de la varful "
           << (s + 1) << " sunt:\n";
      cout << Dijkstra( G, m, s ) << '\n';
    }

    return 0;
  }
```

6.10 Euristica greedy

Pentru anumite probleme, se poate accepta utilizarea unor algoritmi despre care nu se știe dacă furnizează soluția optimă, dar care furnizează rezultate "acceptabile", sunt mai ușor de implementat și mai eficienți decât algoritmii care dau soluția optimă. Un astfel de algoritm se numește *euristic*.

Una din ideile frecvent utilizate în elaborarea algoritmilor euristici constă în descompunerea procesului de căutare a soluției optime în mai multe subprocese succesive, fiecare din aceste subprocese constând dintr-o optimizare. O astfel de strategie nu poate conduce întotdeauna la o soluție optimă, deoarece alegerea unei soluții optime la o anumită etapă poate împiedica atingerea în final a unei soluții optime a întregii probleme; cu alte cuvinte, optimizarea locală nu implică, în general, optimizarea globală. Regăsim, de fapt, principiul care stă la baza metodei greedy. Un algoritm greedy, despre care nu se poate demonstra că furnizează soluția optimă, este un algoritm euristic.

Vom da două exemple de utilizare a algoritmilor greedy euristici.

6.10.1 Colorarea unui graf

Fie $G = <V, M>$ un graf neorientat, ale cărui vârfuri trebuie colorate astfel încât oricare două vârfuri adiacente să fie colorate diferit. Problema este de a obține o colorare cu un număr minim de culori.

Folosim următorul algoritm greedy: alegem o culoare și un vârf arbitrar de pornire, apoi considerăm vârfurile rămase, încercând să le colorăm, fără a

schimba culoarea. Când nici un vârf nu mai poate fi colorat, schimbăm culoarea și vârful de start, repetând procedeul.

Dacă în graful din Figura 6.6 pornim cu vârful 1 și îl colorăm în roșu, mai putem colora tot în roșu vârfurile 3 și 4. Apoi, schimbăm culoarea și pornim cu vârful 2, colorându-l în albastru. Mai putem colora cu albastru și vârful 5. Deci, ne-au fost suficiente două culori. Dacă colorăm vârfurile în ordinea 1, 5, 2, 3, 4, atunci se obține o colorare cu trei culori.

Rezultă că, prin metoda greedy, nu obținem decât o soluție euristică, care nu este în mod necesar soluția optimă a problemei. De ce suntem atunci interesați într-o astfel de rezolvare? Toți algoritmii cunoscuți, care rezolvă optim această problemă, sunt exponențiali, deci, practic, nu pot fi folosiți pentru cazuri mari. Algoritmul greedy euristic propus furnizează doar o soluție "acceptabilă", dar este simplu și eficient.

Un caz particular al problemei colorării unui graf corespunde celebrei *probleme a colorării hărților*: o hartă oarecare trebuie colorată cu un număr minim de culori, astfel încât două țări cu frontieră comună să fie colorate diferit. Dacă fiecărui vârf îi corespunde o țară, iar două vârfuri adiacente reprezintă țări cu frontieră comună, atunci hărții îi corespunde un graf *planar*, adică un graf care poate fi desenat în plan fără ca două muchii să se intersecteze. Celebritatea problemei constă în faptul că, în toate exemplele întâlnite, colorarea s-a putut face cu cel mult 4 culori. Aceasta în timp ce, teoretic, se putea demonstra că pentru o hartă oarecare este nevoie de cel mult 5 culori. Recent* s-a demonstrat pe calculator faptul că orice hartă poate fi colorată cu cel mult 4 culori. Este prima demonstrare pe calculator a unei teoreme importante.

Problema colorării unui graf poate fi interpretată și în contextul planificării unor activități. De exemplu, să presupunem că dorim să executăm simultan o mulțime de activități, în cadrul unor săli de clasă. În acest caz, vârfurile grafului reprezintă activități, iar muchiile unesc activitățile incompatibile. Numărul minim de culori necesare pentru a colora graful corespunde numărului minim de săli necesare.

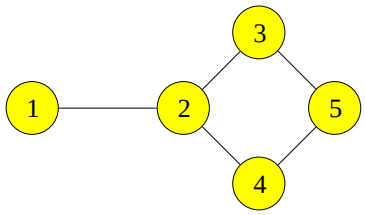


Figura 6.6 Un graf care va fi colorat.

* K. Appel și W. Haken, în 1976.

La:	2	3	4	5	6
De la:					
1	3	10	11	7	25
2		6	12	8	26
3			9	4	20
4				5	15
5					18

Tabelul 6.4 Matricea distanțelor pentru problema comis-voiajorului.

6.10.2 Problema comis-voiajorului

Se cunosc distanțele dintre mai multe orașe. Un comis-voiajor pleacă dintr-un oraș și dorește să se întoarcă în același oraș, după ce a vizitat fiecare din celelalte orașe exact o dată. Problema este de a minimiza lungimea drumului parcurs. Și pentru această problemă, toți algoritmii care găsesc soluția optimă sunt exponențiali.

Problema poate fi reprezentată printr-un graf neorientat, în care oricare două vârfuri diferite ale grafului sunt unite între ele printr-o muchie, de lungime nenegativă. Căutăm un ciclu de lungime minimă, care să se închidă în vârful inițial și care să treacă prin toate vârfurile grafului.

Conform strategiei greedy, vom construi ciclul pas cu pas, adăugând la fiecare iterație cea mai scurtă muchie disponibilă cu următoarele proprietăți:

- nu formează un ciclu cu muchiile deja selectate (exceptând pentru ultima muchie aleasă, care completează ciclul)
- nu există încă două muchii deja selectate, astfel încât cele trei muchii să fie incidente în același vârf

De exemplu, pentru șase orașe a căror matrice a distanțelor este dată în Tabelul 6.4, muchiile se aleg în ordinea: {1, 2}, {3, 5}, {4, 5}, {2, 3}, {4, 6}, {1, 6} și se obține ciclul (1, 2, 3, 5, 4, 6, 1) de lungime 58. Algoritmul greedy nu a găsit ciclul optim, deoarece ciclul (1, 2, 3, 6, 4, 5, 1) are lungimea 56.

6.11 Exerciții

6.1 Presupunând că există monezi de:

i) 1, 5, 12 și 25 de unități, găsiți un contraexemplu pentru care algoritmul greedy nu găsește soluția optimă;

ii) 10 și 25 de unități, găsiți un contraexemplu pentru care algoritmul greedy nu găsește nici o soluție cu toate că există soluție.

6.2 Presupunând că există monezi de:

$$k^0, k^1, \ldots, k^{n-1}$$

unități, pentru $k \in \mathbf{N}$, $k > 1$ oarecare, arătați că metoda greedy dă mereu soluția optimă. Considerați că n este un număr finit și că din fiecare tip de monedă există o cantitate nelimitată.

6.3 Pe o bandă magnetică sunt n programe, un program i de lungime l_i fiind apelat cu probabilitatea p_i, $1 \le i \le n$, $p_1+p_2+\ldots+p_n = 1$. Pentru a citi un program, trebuie să citim banda de la început. În ce ordine să memorăm programele pentru a minimiza timpul mediu de citire a unui program oarecare?

Indicație: Se pun în ordinea descrescătoare a rapoartelor p_i / l_i.

6.4 Analizați eficiența algoritmului greedy care planifică ordinea clienților într-o stație de servire, minimizând timpul mediu de așteptare.

6.5 Pentru un text format din n litere care apar cu frecvențele $f_1, f_2, \ldots, f_n$, demonstrați că arborele de codificare Huffman minimizează lungimea externă ponderată pentru toți arborii de codificare cu vârfurile terminale având valorile $f_1, f_2, \ldots, f_n$.

6.6 Câți biți ocupă textul “ABRACADABRA” după codificarea Huffman?

6.7 Ce se întâmplă când facem o codificare Huffman a unui text binar? Ce se întâmplă când facem o codificare Huffman a unui text format din litere care au aceeași frecvență?

6.8 Elaborați algoritmul de compactare Huffman a unui șir de caractere.

6.9 Elaborați algoritmul de decompactare a unui șir de caractere codificat prin codul Huffman, presupunând că se cunosc caracterele și codificarea lor. Folosiți proprietatea că acest cod este de tip prefix.

6.10 Pe lângă codul Huffman, vom considera aici și un alt cod celebru, care nu se obține însă printr-o metodă greedy, ci printr-un algoritm recursiv.

Un *cod Gray* este o secvență de 2^n elemente astfel încât:

i) fiecare element este un șir de n biți

ii) oricare două elemente sunt diferite

iii) oricare două elemente consecutive diferă exact printr-un bit (primul element este considerat succesorul ultimului element)

Se observă că un cod Gray nu este de tip prefix. Elaborați un algoritm recursiv pentru a construi codul Gray pentru orice n dat. Gândiți-vă cum ați putea utiliza un astfel de cod.

Indicație: Pentru $n = 1$ putem folosi secvența (0, 1). Presupunem că avem un cod Gray pentru $n–1$, unde $n > 1$. Un cod Gray pentru n poate fi construit prin concatenarea a două subsecvențe. Prima se obține prefixând cu 0 fiecare element al codului Gray pentru $n–1$. A doua se obține citind în ordine inversă codul Gray pentru $n–1$ și prefixând cu 1 fiecare element rezultat.

6.11 Demonstrați că graful parțial definit ca arbore parțial de cost minim este un arbore.

Indicație: Arătați că orice graf conex cu n vârfuri are cel puțin $n–1$ muchii și revedeți Exercițiul 3.2.

6.12 Dacă în algoritmul lui Kruskal reprezentăm graful nu printr-o listă de muchii, ci printr-o matrice de adiacență, care conține costurile muchiilor, ce se poate spune despre timp?

6.13 Ce se întâmplă dacă rulăm algoritmul *i) Kruskal*, *ii) Prim* pe un graf neconex?

6.14 Ce se întâmplă în cazul algoritmului: *i) Kruskal*, *ii) Prim* dacă permitem muchiilor să aibă cost negativ?

6.15 Să presupunem că am găsit arborele parțial de cost minim al unui graf G. Elaborați un algoritm de actualizare a arborelui parțial de cost minim, după ce am adăugat în G un nou vârf, împreună cu muchiile incidente lui. Analizați algoritmul obținut.

6.16 În graful din Figura 6.5, găsiți pe unde trec cele mai scurte drumuri de la vârful 1 către toate celelalte vârfuri.

6.17 Scrieți algoritmul greedy pentru colorarea unui graf și analizați eficiența lui.

6.18 Ce se întâmplă cu algoritmul greedy din problema comis-voiajorului dacă admitem că pot exista două orașe fără legătură directă între ele?

6.19 Scrieți algoritmul greedy pentru problema comis-voiajorului și analizați eficiența lui.

6.20 Într-un graf orientat, un drum este *hamiltonian* dacă trece exact o dată prin fiecare vârf al grafului, fără să se întoarcă în vârful inițial. Fie G un graf orientat, cu proprietatea că între oricare două vârfuri există cel puțin o muchie. Arătați că în G există un drum hamiltonian și elaborați algoritmul care găsește acest drum.

6.21 Este cunoscut că orice număr natural i poate fi descompus în mod unic într-o sumă de termeni ai șirului lui Fibonacci (teorema lui Zeckendorf). Dacă prin $k >> m$ notăm $k \geq m+2$, atunci

$$i = f_{k_1} + f_{k_2} + \ldots f_{k_r}$$

unde

$$k_1 >> k_2 >> \ldots >> k_r >> 0$$

În acestă *reprezentare Fibonacci* a numerelor, singura valoare posibilă pentru f_{k_1} este cel mai mare termen din șirul lui Fibonacci pentru care $f_{k_1} \leq i$; singura valoare posibilă pentru f_{k_2} este cel mai mare termen pentru care $f_{k_2} \leq i - f_{k_1}$ etc. Reprezentarea Fibonacci a unui număr nu conține niciodată doi termeni consecutivi ai șirului lui Fibonacci.

Pentru $0 \leq i \leq f_n - 1$, $n \geq 3$, numim *codificarea Fibonacci* de ordinul n al lui i secvența de biți $b_{n-1}, b_{n-2}, \ldots, b_2$, unde

$$i = \sum_{j=2}^{n-1} b_j f_j$$

este reprezentarea Fibonacci a lui i. De exemplu, pentru $i = 6$, codificarea de ordinul 6 este 1001, iar codificarea de ordinul 7 este 01001. Se observă că în codificarea Fibonacci nu apar doi de 1 consecutiv.

Dați un algoritm pentru determinarea codificării Fibonacci de ordinul n al lui i, unde n și i sunt oarecare.

6.22 *Codul Fibonacci* de ordinul n, $n \geq 2$, este secvența C_n a celor f_n codificări Fibonacci de ordinul n ale lui i, atunci când i ia toate valorile $0 \leq i \leq f_n - 1$. De exemplu, dacă notăm cu λ șirul nul, obținem: $C_2 = (\lambda)$, $C_3 = (0, 1)$, $C_4 = (00, 01, 10)$, $C_5 = (000, 001, 010, 100, 101)$ etc. Elaborați un algoritm recursiv care construiește codul Fibonacci pentru orice n dat. Gândiți-vă cum ați putea utiliza un astfel de cod.

Indicație: Arătați că putem construi codul Fibonacci de ordinul n, $n \geq 4$, prin concatenarea a două subsecvențe. Prima subsecvență se obține prefixând cu 0 fiecare codificare din C_{n-1}. A doua subsecvență se obține prefixând cu 10 fiecare codificare din C_{n-2}.

7. Algoritmi divide et impera

7.1 Tehnica divide et impera

Divide et impera este o tehnică de elaborare a algoritmilor care constă în:

- Descompunerea cazului ce trebuie rezolvat într-un număr de subcazuri mai mici ale aceleiași probleme.
- Rezolvarea succesivă și independentă a fiecăruia din aceste subcazuri.
- Recompunerea subsoluțiilor astfel obținute pentru a găsi soluția cazului inițial.

Să presupunem că avem un algoritm *A* cu timp pătratic. Fie *c* o constantă, astfel încât timpul pentru a rezolva un caz de mărime *n* este $t_A(n) \le cn^2$. Să presupunem că este posibil să rezolvăm un astfel de caz prin descompunerea în trei subcazuri, fiecare de mărime $\lceil n/2 \rceil$. Fie *d* o constantă, astfel încât timpul necesar pentru descompunere și recompunere este $t(n) \le dn$. Folosind vechiul algoritm și ideea de descompunere-recompunere a subcazurilor, obținem un nou algoritm *B*, pentru care:

$$t_B(n) = 3t_A(\lceil n/2 \rceil)+t(n) \le 3c((n+1)/2)^2+dn = 3/4cn^2+(3/2+d)n+3/4c$$

Termenul $3/4cn^2$ domină pe ceilalți când *n* este suficient de mare, ceea ce înseamnă că algoritmul *B* este în esență cu 25% mai rapid decât algoritmul *A*. Nu am reușit însă să schimbăm ordinul timpului, care rămâne pătratic.

Putem să continuăm în mod recursiv acest procedeu, împărțind subcazurile în subsubcazuri etc. Pentru subcazurile care nu sunt mai mari decât un anumit prag n_0, vom folosi tot algoritmul *A*. Obținem astfel algoritmul *C*, cu timpul

$$t_C(n) = \begin{cases} t_A(n) & \text{pentru } n \le n_0 \\ 3t_C(\lceil n/2 \rceil) + t(n) & \text{pentru } n > n_0 \end{cases}$$

Conform rezultatelor din Secțiunea 5.3.5, $t_C(n)$ este în ordinul lui $n^{\lg 3}$. Deoarece $\lg 3 \cong 1{,}59$, înseamnă că de această dată am reușit să îmbunătățim ordinul timpului.

Iată o descriere generală a metodei divide et impera:

```
function divimp(x)
    {returnează o soluție pentru cazul x}
    if x este suficient de mic then return adhoc(x)
    {descompune x în subcazurile x_1, x_2, ..., x_k}
    for i ← 1 to k do y_i ← divimp(x_i)
    {recompune y_1, y_2, ..., y_k în scopul obținerii soluției y pentru x}
    return y
```

unde *adhoc* este subalgoritmul de bază folosit pentru rezolvarea micilor subcazuri ale problemei în cauză (în exemplul nostru, acest subalgoritm este A).

Un algoritm divide et impera trebuie să evite descompunerea recursivă a subcazurilor "suficient de mici", deoarece, pentru acestea, este mai eficientă aplicarea directă a subalgoritmului de bază. Ce înseamnă însă "suficient de mic"?

În exemplul precedent, cu toate că valoarea lui n_0 nu influențează ordinul timpului, este influențată însă constanta multiplicativă a lui $n^{\lg 3}$, ceea ce poate avea un rol considerabil în eficiența algoritmului. Pentru un algoritm divide et impera oarecare, chiar dacă ordinul timpului nu poate fi îmbunătățit, se dorește optimizarea acestui prag în sensul obținerii unui algoritm cât mai eficient. Nu există o metodă teoretică generală pentru aceasta, pragul optim depinzând nu numai de algoritmul în cauză, dar și de particularitatea implementării. Considerând o implementare dată, pragul optim poate fi determinat empiric, prin măsurarea timpului de execuție pentru diferite valori ale lui n_0 și cazuri de mărimi diferite.

În general, se recomandă o metodă hibridă care constă în: *i)* determinarea teoretică a formei ecuațiilor recurente; *ii)* găsirea empirică a valorilor constantelor folosite de aceste ecuații, în funcție de implementare.

Revenind la exemplul nostru, pragul optim poate fi găsit rezolvând ecuația

$$t_A(n) = 3t_A(\lceil n/2 \rceil) + t(n)$$

Empiric, găsim $n_0 \cong 67$, adică valoarea pentru care nu mai are importanță dacă aplicăm algoritmul A în mod direct, sau dacă continuăm descompunerea. Cu alte cuvinte, atâta timp cât subcazurile sunt mai mari decât n_0, este bine să continuăm descompunerea. Dacă continuăm însă descompunerea pentru subcazurile mai mici decât n_0, eficiența algoritmului scade.

Observăm că metoda divide et impera este prin definiție recursivă. Uneori este posibil să eliminăm recursivitatea printr-un ciclu iterativ. Implementată pe o mașină convențională, versiunea iterativă *poate fi* ceva mai rapidă (în limitele unei constante multiplicative). Un alt avantaj al versiunii iterative ar fi faptul că economisește spațiul de memorie. Versiunea recursivă folosește o stivă necesară

memorării apelurilor recursive. Pentru un caz de mărime n, numărul apelurilor recursive este de multe ori în $\Omega(\log n)$, uneori chiar în $\Omega(n)$.

7.2 Căutarea binară

Căutarea binară este cea mai simplă aplicație a metodei divide et impera, fiind cunoscută încă înainte de apariția calculatoarelor. În esență, este algoritmul după care se caută un cuvînt într-un dicționar, sau un nume în cartea de telefon.

Fie $T[1 .. n]$ un tablou ordonat crescător și x un element oarecare. Problema constă în a-l găsi pe x în T, iar dacă nu se află acolo în a găsi poziția unde poate fi inserat. Căutăm deci indicele i astfel încât $1 \le i \le n$ și $T[i] \le x < T[i+1]$, cu convenția $T[0] = -\infty$, $T[n+1] = +\infty$. Cea mai evidentă metodă este *căutarea secvențială*:

```
function sequential(T[1 .. n], x)
    {caută secvențial pe x în tabloul T }
    for i ← 1 to n do
        if T[i] > x then return i−1
    return n
```

Algoritmul necesită un timp în $\Theta(1+r)$, unde r este indicele returnat; aceasta înseamnă $\Theta(1)$ pentru cazul cel mai favorabil și $\Theta(n)$ pentru cazul cel mai nefavorabil. Dacă presupunem că elementele lui T sunt distincte, că x este un element al lui T și că se află cu probabilitate egală în oricare poziție din T, atunci bucla **for** se execută în medie de $(n^2+3n-2)/2n$ ori. Timpul este deci în $\Theta(n)$ și pentru cazul mediu.

Pentru a mări viteza de căutare, metoda divide et impera sugerează să-l căutăm pe x fie în prima jumătate a lui T, fie în cea de-a doua. Comparându-l pe x cu elementul din mijlocul tabloului, putem decide în care dintre jumătăți să căutăm. Repetând recursiv procedeul, obținem următorul algoritm de *căutare binară*:

```
function binsearch(T[1 .. n], x)
    {caută binar pe x în tabloul T}
    if n = 0 or x < T[1] then return 0
    return binrec(T[1 .. n], x)
```

```
function binrec(T[i .. j], x)
    {caută binar pe x în subtabloul T[i .. j]; această procedură
     este apelată doar când T[i] ≤ x < T[ j+1] şi i ≤ j}
    if i = j then return i
    k ← (i+j+1) div 2
    if x < T[k]   then   return binrec(T[i .. k–1], x)
                  else   return binrec(T[k .. j], x)
```

Algoritmul *binsearch* necesită un timp în Θ(log *n*), indiferent de poziția lui *x* în *T* (demonstrați acest lucru, revăzând Secțiunea 5.3.5). Procedura *binrec* execută doar un singur apel recursiv, în funcție de rezultatul testului "*x* < *T*[*k*]". Din această cauză, căutarea binară este, mai curând, un exemplu de simplificare, decât de aplicare a tehnicii divide et impera.

Iată și versiunea iterativă a acestui algoritm:

```
function iterbin1(T[1 .. n], x)
    {căutare binară iterativă}
    if n = 0 or x < T[1] then return 0
    i ← 1; j ← n
    while i < j do
        {T[i] ≤ x < T[ j+1]}
        k ← (i+j+1) div 2
        if x < T[k]   then   j ← k–1
                      else   i ← k
    return i
```

Acest algoritm de căutare binară pare ineficient în următoarea situație: dacă la un anumit pas avem *x* = *T*[*k*], se continuă totuși căutarea. Următorul algoritm evită acest inconvenient, oprindu-se imediat ce găsește elementul căutat.

```
function iterbin2(T[1 .. n], x)
    {variantă a căutării binare iterative}
    if n = 0 or x < T[1] then return 0
    i ← 1; j ← n
    while i < j do
        {T[i] ≤ x < T[ j+1]}
        k ← (i+j) div 2
        case   x < T[k]: j ← k–1
               x ≥ T[k+1]:  i ← k+1
               otherwise:   i, j ← k
    return i
```

Timpul pentru *iterbin*1 este în Θ(log *n*). Algoritmul *iterbin*2 necesită un timp care depinde de poziția lui *x* în *T*, fiind în Θ(1), Θ(log *n*), Θ(log *n*) pentru cazurile cel mai favorabil, mediu și respectiv, cel mai nefavorabil.

Care din acești doi algoritmi este oare mai eficient? Pentru cazul cel mai favorabil, *iterbin*2 este, evident, mai bun. Pentru cazul cel mai nefavorabil, ordinul timpului este același, numărul de executări ale buclei **while** este același, dar durata unei bucle **while** pentru *iterbin*2 este ceva mai mare; deci *iterbin*1 este preferabil, având constanta multiplicativă mai mică. Pentru cazul mediu, compararea celor doi algoritmi este mai dificilă: ordinul timpului este același, o buclă **while** în *iterbin*1 durează în medie mai puțin decât în *iterbin*2, în schimb *iterbin*1 execută în medie mai multe bucle **while** decât *iterbin*2.

7.3 Mergesort (sortarea prin interclasare)

Fie $T[1 .. n]$ un tablou pe care dorim să-l sortăm crescător. Prin tehnica divide et impera putem proceda astfel: separăm tabloul T în două părți de mărimi cât mai apropiate, sortăm aceste părți prin apeluri recursive, apoi interclasăm soluțiile pentru fiecare parte, fiind atenți să păstrăm ordonarea crescătoare a elementelor. Obținem următorul algoritm:

```
procedure mergesort(T[1 .. n])
    {sortează în ordine crescătoare tabloul T}
    if n este mic
        then  insert(T)
        else  arrays U[1 .. n div 2], V[1 .. (n+1) div 2]
              U ← T[1 .. n div 2]
              V ← T[1 + (n div 2) .. n]
              mergesort(U); mergesort(V)
    merge(T, U, V)
```

unde *insert*(*T*) este algoritmul de sortare prin inserție cunoscut, iar *merge*(*T*, *U*, *V*) interclasează într-un singur tablou sortat *T* cele două tablouri deja sortate *U* și *V*.

Algoritmul *mergesort* ilustrează perfect principiul divide et impera: pentru *n* având o valoare mică, nu este rentabil să apelăm recursiv procedura *mergesort*, ci este mai bine să efectuăm sortarea prin inserție. Algoritmul *insert* lucrează foarte bine pentru $n \leq 16$, cu toate că, pentru o valoare mai mare a lui *n*, devine neconvenabil. Evident, se poate concepe un algoritm mai puțin eficient, care să meargă până la descompunerea totală; în acest caz, mărimea stivei este în $\Theta(\log n)$.

Spațiul de memorie necesar pentru tablourile auxiliare *U* și *V* este în $\Theta(n)$. Mai precis, pentru a sorta un tablou de $n = 2^k$ elemente, presupunând că descompunerea este totală, acest spațiu este de

$$2(2^{k-1} + 2^{k-2} + \ldots + 2 + 1) = 2 \cdot 2^k = 2n$$

elemente.

Putem considera (conform Exercițiului 7.7) că algoritmul *merge*(*T*, *U*, *V*) are timpul de execuție în $\Theta(\#U + \#V)$, indiferent de ordonarea elementelor din *U* și *V*. Separarea lui *T* în *U* și *V* necesită tot un timp în $\Theta(\#U + \#V)$. Timpul necesar algoritmului *mergesort* pentru a sorta orice tablou de *n* elemente este atunci $t(n) \in t(\lfloor n/2 \rfloor)+t(\lceil n/2 \rceil)+\Theta(n)$. Această ecuație, pe care am analizat-o în Secțiunea 5.1.2, ne permite să conchidem că timpul pentru *mergesort* este în $\Theta(n \log n)$. Să reamintim timpii celorlalți algoritmi de sortare, algoritmi analizați în Capitolul 5: pentru cazul mediu și pentru cazul cel mai nefavorabil *insert* și *select* necesită un timp în $\Theta(n^2)$, iar *heapsort* un timp în $\Theta(n \log n)$.

În algoritmul *mergesort*, suma mărimilor subcazurilor este egală cu mărimea cazului inițial. Această proprietate nu este în mod necesar valabilă pentru algoritmii divide et impera. Oare de ce este însă important ca subcazurile să fie de mărimi cât mai egale? Dacă în *mergesort* îl separăm pe *T* în tabloul *U* având $n-1$ elemente și tabloul *V* având un singur element, se obține (Exercițiul 7.9) un nou timp de execuție, care este în $\Theta(n^2)$. Deducem de aici că este esențial ca subcazurile să fie de mărimi cât mai apropiate (sau, alfel spus, subcazurile să fie cât mai *echilibrate*).

7.4 Mergesort în clasele `tablou<T>` și `lista<E>`

7.4.1 O soluție neinspirată

Deși eficient în privința timpului, algoritmul de sortare prin interclasare are un handicap important în ceea ce privește memoria necesară. Într-adevăr, orice tablou de *n* elemente este sortat într-un timp în $\Theta(n \log n)$, dar utilizând un spațiu suplimentar de memorie* de $2n$ elemente. Pentru a reduce consumul de memorie, în implementarea acestui algoritm nu vom utiliza variabilele intermediare `U` și `V` de tip `tablou<T>`, ci o unică zonă de auxiliară de *n* elemente.

Convenim să implementăm procedura *mergesort* din Secțiunea 7.3 ca membru `private` al clasei parametrice `tablou<T>`. Invocarea acestei proceduri se va realiza prin funcția membră

---

* Spațiul suplimentar utilizat de algoritmul *mergesort* poate fi independent de numărul elementelor tabloului de sortat. Detaliile de implementare a unei astfel de strategii se găsesc în D. E. Knuth, "*Tratat de programarea calculatoarelor. Sortare și căutare*", Secțiunea 5.2.4.

```
template <class T>
tablou<T>& tablou<T>::sort( ) {
  T *aux = new T[ d ];    // alocarea zonei de interclasare
  mergesort( 0, d, aux ); //       si sortarea propriu-zisa
  delete [ ] aux;         // eliberarea zonei alocate

  return *this;
}
```

Am preferat această manieră de "încapsulare" din următoarele două motive:

- Alocarea și eliberarea spațiului suplimentar necesar interclasării se face o singură dată, înainte și după terminarea sortării. Funcția `mergesort()`, ca funcție recursivă, nu poate avea controlul asupra alocării și eliberării acestei zone.
- Algoritmul *mergesort* are trei parametri care pot fi ignorați la apelarea funcției de sortare. Aceștia sunt: adresa zonei suplimentare de memorie și cei doi indici prin care se încadrează elementele de sortat din tablou.

După cum se poate vedea în Exercițiul 7.7, implementarea interclasării se simplifică mult prin utilizarea unor valori "santinelă" în tablourile de interclasat. Funcția `mergesort()`:

```
template <class T>
void tablou<T>::mergesort( int st, int dr, T *x ) {
  if ( dr - st > 1 ) {
    // mijlocul intervalului
    int m = ( st + dr ) / 2;

    // sortarea celor doua parti
    mergesort( st, m );
    mergesort( m, dr );

    // pregatirea zonei x pentru interclasare
    int k = st;
    for ( int i = st; i < m; ) x[ i++ ] = a[ k++ ];
    for ( int j = dr; j > m; ) x[ --j ] = a[ k++ ];

    // interclasarea celor doua parti din x in zona a
    i = st; j = dr - 1;
    for ( k = st; k < dr; k++ )
    a[ k ] = x[ j ] > x[ i ]? x[ i++ ]: x[ j-- ];
  }
}
```

se adaptează surprinzător de simplu la utilizarea "santinelelor". Nu avem decât să transferăm în zona auxiliară cele două jumătăți deja sortate, astfel încât valorile maxime să fie la mijlocul acestei zone. Altfel spus, prima jumătate va fi transferată crescător, iar cea de-a doua descrescător, în continuarea primei jumătăți. Începând interclasarea cu valorile minime, valoarea maximă din fiecare jumătate este santinelă pentru cealaltă jumătate.

Sortarea prin interclasare prezintă un avantaj foarte important față de alte metode de sortare deoarece elementele de sortat sunt parcurse secvențial, element după element. Din acest motiv, metoda este potrivită pentru sortarea fișierelor sau listelor. De exemplu, procedura de sortare prin interclasare a obiectelor de tip `lista<E>`

```
template <class E>
lista<E>& lista<E>::sort() {
  if ( head )
    head = mergesort( head );

  return *this;
}
```

rearanjează nodurile în ordinea crescătoare a cheilor, fără a folosi noduri sau liste temporare. Prețul în spațiu suplimentar de memorie este totuși plătit, deoarece orice listă înlănțuită necesită memorie în ordinul numărului de elemente pentru realizarea înlănțuirii.

Conform algoritmului *mergesort*, lista se împarte în două parți egale, iar după sortarea fiecăreia se realizează interclasarea. Împărțirea listei în cele două părți egale nu se poate realiza direct, ca în cazul tablourilor, ci în mai mulți pași. Astfel, vom parcurge lista până la sfârșit, pentru a putea determina elementul din mijloc. Apoi stabilim care este elementul din mijloc și, în final, izolăm cele două părți, fiecare în câte o listă. În funcția `mergesort()`:

```
template <class E>
nod<E>* mergesort ( nod<E> *c ) {
  if ( c && c->next ) {
    // sunt cel putin doua noduri in lista
    nod<E> *a = c, *b;

    for ( b = c->next; b; a = a->next )
      if ( b->next ) b = b->next->next;
      else           break;
    b = a->next; a->next = 0;
    return merge( mergesort( c ), mergesort( b ) );
  }
  else
    // lista contine cel mult un nod
    return c;
}
```

împărțirea listei se realizează printr-o singură parcurgere, dar cu două adrese de noduri, `a` și `b`. Principiul folosit este următorul: dacă `b` înaintează în parcurgerea listei de două ori mai repede decât `a`, atunci când `b` a ajuns la ultimul nod, `a` este la nodul de mijloc al listei.

Spre deosebire de algoritmul *mergesort*, sortarea listelor prin interclasare nu deplasează valorile de sortat. Funcția `merge()` interclasează listele de la adresele `a` și `b` prin simpla modificare a legăturilor nodurilor.

```
template <class E>
nod<E>* merge( nod<E> *a, nod<E> *b ) {
  nod<E> *head;           // primul nod al listei interclasate

  if ( a && b )
    // ambele liste sunt nevide;
    // stabilim primul nod din lista interclasata
    if ( a->val > b->val ) { head = b; b = b->next; }
    else                   { head = a; a = a->next; }
  else
    // cel putin una din liste este vida;
    // nu avem ce interclasa
    return a? a: b;

  // interclasarea propriu-zisa
  nod<E> *c = head;       // ultimul nod din lista interclasata
  while ( a && b )
    if ( a->val > b->val ) { c->next = b; c = b; b = b->next; }
    else                   { c->next = a; c = a; a = a->next; }

  // cel putin una din liste s-a epuizat
  c->next = a? a: b;

  // se returneaza primul nod al listei interclasate
  return head;
}
```

Funcția de sortare `mergesort()`, împreună cu cea de interclasare `merge()`, lucrează exclusiv asupra nodurilor. Deoarece aceste funcții sunt invocate doar la nivel de listă, ele nu sunt membre în clasa `nod<E>`, ci doar `friend` față de această clasă. Încapsularea lor este realizată prin mecanismul standard al limbajului C++. Deși aceste funcții aparțin domeniului global, ele nu pot fi invocate de aici datorită obiectelor de tip `nod<E>`, obiecte accesibile doar din domeniul clasei `lista<E>`. Această manieră de încapsulare nu este complet sigură, deoarece, chiar dacă nu putem manipula obiecte de tip `nod<E>`, totuși putem lucra cu adrese de `nod<E>`. De exemplu, funcția

```
void f( ) {
  mergesort( (nod<int> *)0 );
}
```

“trece” de compilare, dar efectele ei la rularea programului sunt imprevizibile.

Prezența funcțiilor de sortare în `tablou<T>` și `lista<E>` (de fapt și în `nod<E>`) impune completarea claselor `T` și `E` cu operatorul de comparare `>`. Orice tentativă de a defini (atenție, de a *defini* și nu de a *sorta*) obiecte de tip `tablou<T>` sau `lista<E>` este semnalată ca eroare de compilare, dacă tipurile `T` sau `E` nu au definit acest operator. Situația apare, deoarece generarea unei clase parametrice implică generarea tuturor funcțiilor membre. Deci, chiar dacă nu invocăm funcția

de sortare pentru tipul tablou<T>, ea este totuși generată, iar generarea ei necesită operatorul de comparare al tipului T.

De exemplu, pentru a putea lucra cu liste de muchii, lista<muchie>, sau tablouri de tablouri, tablou< tablou<T> >, vom implementa operatorii de comparare pentru clasa muchie și clasa tablou<T>. Muchiile sunt comparate în funcție de costul lor, dar cum vom proceda cu tablourile? O soluție este de a lucra conform ordinii lexicografice, adică de a aplica aceeași metodă care se aplică la ordonarea numelor în cartea de telefoane, sau în catalogul școlar:

```
template <class T>
operator > ( const tablou<T>& a, const tablou<T>& b ) {
  // minumul elementelor
  int as = a.size( ), bs = b.size( );
  int n = as < bs? as: bs;

  // comparam pana la prima diferenta
  for ( int i = 0; i < n; i++ )
    if ( a[ i ] != b[ i ] ) return a[ i ] > b[ i ];

  // primele n elemente sunt identice
  return as > bs;
}
```

Atunci când operatorii de comparare nu prezintă interes, sau nu pot fi definiți, îi putem implementa ca funcții inefective. Astfel, dacă avem nevoie de un tablou de liste sau de o listă de liste asupra cărora nu vom aplica operații de sortare, va trebui să definim operatorii inefectivi:

```
template <class E>
operator >( const lista<E>&, const lista<E>& ) {
  return 1;
}
```

În concluzie, extinderea claselor tablou<T> și lista<E> cu funcțiile de sortare nu menține compatibilitatea acestor clase față de aplicațiile dezvoltate până acum. Oricând este posibil ca recompilarea unei aplicații în care se utilizează, de exemplu, tablouri sau liste cu elemente de tip XA, XB etc, să devină un coșmar, deoarece, chiar dacă nu are nici un sens, trebuie să completăm fiecare clasă XA, XB etc, cu operatorul de comparare >.

Programarea orientată pe obiect se folosește tocmai pentru a evita astfel de situații, nu pentru a le genera.

7.4.2 Tablouri sortabile și liste sortabile

Sortarea este o operație care completează facilitățile clasei `tablou<T>`, fără a exclude utilizarea acestei clase pentru tablouri nesortabile. Din acest motiv, funcțiile de sortare nu pot fi funcții membre în clasa `tablou<T>`.

O soluție posibilă de încapsulare a sortării este de a construi, prin derivare publică din `tablou<T>`, subtipul `tablouSortabil<T>`, care să conțină tot ceea ce este necesar pentru sortare. Mecanismului standard de conversie, de la tipul derivat public la tipul de bază, permite ca un `tablouSortabil<T>` să poată fi folosit oricând în locul unui `tablou<T>`.

În continuare, vom prezenta o altă variantă de încapsulare, mai puțin clasică, prin care atributul "sortabil" este considerat doar în momentul invocării funcției de sortatre, nu apriori, prin definirea obiectului ca "sortabil".

Sortarea se invocă prin funcția

```
template <class T>
tablou<T>& mergesort( tablou<T>& t ) {
  ( tmsort<T> )t;
  return t;
}
```

care constă în conversia tabloului `t` la tipul `tmsort<T>`. Clasa `tmsort<T>` încapsulează absolut toate detaliile sortării. Fiind vorba de sortarea prin interclasare, detaliile de implementare sunt cele stabilite în Secțiunea 7.4.1.

```
template <class T>
class tmsort {
public:
  tmsort( tablou<T>& );

private:
  T *a;  // adresa zonei de sortat
  T *x;  // zona auxiliara de interclasare

  void mergesort( int, int );
};
```

Sortarea, de fapt transformarea tabloului `t` într-un tablou sortat, este realizată prin constructorul

```
template <class T>
tmsort<T>::tmsort( tablou<T>& t ): a( t.a ) {
  x = new T[ t.size( ) ];    // alocarea zonei de interclasare
  mergesort( 0, t.size( ) ); // sortarea
  delete [ ] x;              // eliberarea zonei alocate
}
```

După cum se observă, în acest constructor se folosește membrul privat `T *a` (adresa zonei alocate elementelor tabloului) din clasa `tablou<T>`. Iată de ce, în clasa `tablou<T>` trebuie făcută o modificare (singura dealtfel): clasa `tmsort<T>` trebuie declarată `friend`.

Funcția `mergesort()` este practic neschimbată:

```
template <class T>
void tmsort<T>::mergesort( int st, int dr ) {
  // ...
  // corpul functiei void mergesort( int, int, T* )
  // din Sectiunea 7.4.1.
  // ...
}
```

Pentru sortarea listelor se procedează analog, transformând implementarea din Secțiunea 7.4.1 în cea de mai jos.

```
template <class E>
lista<E>& mergesort( lista<E>& l ) {
  ( lmsort<E> )l;
  return l;
}

template <class E>
class lmsort {
public:
  lmsort( lista<E>& );

private:
  nod<E>* mergesort( nod<E>* );
  nod<E>* merge( nod<E>*, nod<E>* );
};

template <class E>
lmsort<E>::lmsort( lista<E>& l ) {
  if ( l.head )
    l.head = mergesort( l.head );
}
```

```
template <class E>
nod<E>* lmsort<E>::mergesort ( nod<E> *c ) {
  // ...
  // corpul functiei nod<E>* mergesort( nod<E>* )
  // din Sectiunea 7.4.1.
  // ...
}

template <class E>
nod<E>* lmsort<E>::merge( nod<E> *a, nod<E> *b ) {
  // ...
  // corpul functiei nod<E>* merge( nod<E>*, nod<E>* )
  // din Sectiunea 7.4.1.
  // ...
}
```

Nu uitați de declarația `friend`! Clasa `lmsort<E>` folosește membrii privați atât din clasa `lista<E>`, cât și din clasa `nod<E>`, deci trebuie declarată `friend` în ambele.

7.5 Quicksort (sortarea rapidă)

Algoritmul de sortare *quicksort*, inventat de Hoare în 1962, se bazează de asemenea pe principiul divide et impera. Spre deosebire de *mergesort*, partea nerecursivă a algoritmului este dedicată construirii subcazurilor și nu combinării soluțiilor lor.

Ca prim pas, algoritmul alege un element *pivot* din tabloul care trebuie sortat. Tabloul este apoi partiționat în două subtablouri, alcătuite de-o parte și de alta a acestui pivot în următorul mod: elementele mai mari decât pivotul sunt mutate în dreapta pivotului, iar celelalte elemente sunt mutate în stânga pivotului. Acest mod de partiționare este numit *pivotare*. În continuare, cele două subtablouri sunt sortate în mod independent prin apeluri recursive ale algoritmului. Rezultatul este tabloul complet sortat; nu mai este necesară nici o interclasare. Pentru a echilibra mărimea celor două subtablouri care se obțin la fiecare partiționare, ar fi ideal să alegem ca pivot elementul median. Intuitiv, *mediana* unui tablou T este elementul m din T, astfel încât numărul elementelor din T mai mici decât m este egal cu numărul celor mai mari decât m (o definiție riguroasă a medianei unui tablou este dată în Secțiunea 7.6). Din păcate, găsirea medianei necesită mai mult timp decât merită. De aceea, putem pur și simplu să folosim ca pivot primul element al tabloului. Iată cum arată acest algoritm:

```
procedure quicksort(T[i .. j])
    {sortează în ordine crescătoare tabloul T[i .. j]}
    if j−i este mic
        then  insert(T[i .. j])
        else  pivot(T[i .. j], l)
              {după pivotare, avem:
                  i ≤ k < l   ⇒   T[k] ≤ T[l]
                  l < k ≤ j   ⇒   T[k] > T[l]}
              quicksort(T[i .. l−1])
              quicksort(T[l+1 .. j])
```

Mai rămâne să concepem un algoritm de pivotare cu timp liniar, care să parcurgă tabloul T o singură dată. Putem folosi următoarea tehnică de pivotare: parcurgem tabloul T o singură dată, pornind însă din ambele capete. Încercați să înțelegeți cum funcționează acest algoritm de pivotare, în care $p = T[i]$ este elementul pivot:

```
procedure pivot(T[i .. j], l)
    {permută elementele din  T[i .. j] astfel încât, în final,
     elementele lui T[i .. l−1] sunt ≤ p,
     T[l] = p,
     iar elementele lui T[l+1 .. j] sunt > p}
    p ← T[i]
    k ← i; l ← j+1
    repeat   k ← k+1  until T[k] > p or k ≥ j
    repeat   l ← l−1   until T[l] ≤ p
    while k < l do
        interschimbă T[k] și T[l]
        repeat   k ← k+1  until T[k] > p
        repeat   l ← l−1   until T[l] ≤ p
    {pivotul este mutat în poziția lui finală}
    interschimbă T[i] și T[l]
```

Intuitiv, ne dăm seama că algoritmul *quicksort* este ineficient, dacă se întâmplă în mod sistematic ca subcazurile $T[i \,..\, l-1]$ și $T[l+1 \,..\, j]$ să fie puternic neechilibrate. Ne propunem în continuare să analizăm această situație în mod riguros.

Operația de pivotare necesită un timp în $\Theta(n)$. Fie constanta n_0, astfel încât, în cazul cel mai nefavorabil, timpul pentru a sorta $n > n_0$ elemente prin *quicksort* să fie

$$t(n) \in \Theta(n) + \max\{t(i)+t(n-i-1) \mid 0 \le i \le n-1\}$$

Folosim metoda inducției constructive pentru a demonstra independent că $t \in O(n^2)$ și $t \in \Omega(n^2)$.

Putem considera că există o constantă reală pozitivă c, astfel încât $t(i) \le ci^2+c/2$ pentru $0 \le i \le n_0$. Prin ipoteza inducției specificate parțial, presupunem că $t(i) \le ci^2+c/2$ pentru orice $0 \le i < n$. Demonstrăm că proprietatea este adevărată și pentru n. Avem

$$t(n) \le dn + c + c \max\{i^2+(n-i-1)^2 \mid 0 \le i \le n-1\}$$

d fiind o altă constantă. Expresia $i^2+(n-i-1)^2$ își atinge maximul atunci când i este 0 sau $n-1$. Deci,

$$t(n) \le dn + c + c(n-1)^2 = cn^2 + c/2 + n(d-2c) + 3c/2$$

Dacă luăm $c \ge 2d$, obținem $t(n) \le cn^2+c/2$. Am arătat că, dacă c este suficient de mare, atunci $t(n) \le cn^2+c/2$ pentru orice $n \ge 0$, adică, $t \in O(n^2)$. Analog se arată că $t \in \Omega(n^2)$.

Am arătat, totodată, care este cel mai nefavorabil caz: atunci când, la fiecare nivel de recursivitate, procedura *pivot* este apelată o singură dată. Dacă elementele lui T sunt distincte, cazul cel mai nefavorabil este atunci când inițial tabloul este ordonat crescător sau descrescător, fiecare partiționare fiind total neechilibrată. Pentru acest cel mai nefavorabil caz, am arătat că algoritmul *quicksort* necesită un timp în $\Theta(n^2)$.

Ce se întâmplă însă în cazul mediu? Intuim faptul că, în acest caz, subcazurile sunt suficient de echilibrate. Pentru a demonstra această proprietate, vom arăta că timpul necesar este în ordinul lui $n \log n$, ca și în cazul cel mai favorabil.

Presupunem că avem de sortat n elemente distincte și că inițial ele pot să apară cu probabilitate egală în oricare din cele $n!$ permutări posibile. Operația de pivotare necesită un timp liniar. Apelarea procedurii *pivot* poate poziționa primul element cu probabilitatea $1/n$ în oricare din cele n poziții. Timpul mediu pentru *quicksort* verifică relația

$$t(n) \in \Theta(n) + 1/n \sum_{l=1}^{n} (t(l-1) + t(n-l))$$

Mai precis, fie n_0 și d două constante astfel încât pentru orice $n > n_0$, avem

$$t(n) \le dn + 1/n \sum_{l=1}^{n} (t(l-1) + t(n-l)) = dn + 2/n \sum_{i=0}^{n-1} t(i)$$

Prin analogie cu *mergesort*, este rezonabil să presupunem că $t \in O(n \log n)$ și să aplicăm tehnica inducției constructive, căutând o constantă c, astfel încât $t(n) \le cn \lg n$.

Deoarece $i \lg i$ este o funcție nedescrescătoare, avem

$$\sum_{i=n_0+1}^{n-1} i \lg i \;\le\; \int_{x=n_0+1}^{n} x \lg x \; \mathrm{d}x \;=\; \left[\frac{x^2}{2} \lg x \;-\; \frac{\lg e}{4} x^2 \right]_{x=n_0+1}^{n} \;\le\; \frac{n^2}{2} \lg n \;-\; \frac{\lg e}{4} n^2$$

pentru $n_0 \ge 1$.

Ținând cont de această margine superioară pentru

$$\sum_{i=n_0+1}^{n-1} i \lg i$$

puteți demonstra prin inducție matematică că $t(n) \le cn \lg n$ pentru orice $n > n_0 \ge 1$, unde

$$c \;=\; \frac{2d}{\lg e} \;+\; \frac{4}{(n_0+1)^2 \lg e} \sum_{i=0}^{n_0} t(i)$$

Rezultă că timpul mediu pentru *quicksort* este în $O(n \log n)$. Pe lângă ordinul timpului, un rol foarte important îl are constanta multiplicativă. Practic, constanta multiplicativă pentru *quicksort* este mai mică decât pentru *heapsort* sau *mergesort*. Dacă pentru cazul cel mai nefavorabil se acceptă o execuție ceva mai lentă, atunci, dintre tehnicile de sortare prezentate, *quicksort* este algoritmul preferabil.

Pentru a minimiza șansa unui timp de execuție în $\Omega(n^2)$, putem alege ca pivot mediana șirului $T[i]$, $T[(i+j)$ **div** $2]$, $T[\,j]$. Prețul plătit pentru această modificare este o ușoară creștere a constantei multiplicative.

7.6 Selecția unui element dintr-un tablou

Putem găsi cu ușurință elementul maxim sau minim al unui tablou T. Cum putem determina însă eficient mediana lui T ? Pentru început, să definim formal mediana unui tablou.

Un element m al tabloului $T[1 \,..\, n]$ este mediana lui T, dacă și numai dacă sunt verificate următoarele două relații:

$$\#\{i \in \{1, \ldots, n\} \mid T[i] < m\} < \lceil n/2 \rceil$$

$$\#\{i \in \{1, \ldots, n\} \mid T[i] \le m\} \ge \lceil n/2 \rceil$$

Această definiție ține cont de faptul că n poate fi par sau impar și că elementele din T pot să nu fie distincte. Prima relație este mai ușor de înțeles dacă observăm că

$$\#\{i \in \{1, \ldots, n\} \mid T[i] < m\} = n - \#\{i \in \{1, \ldots, n\} \mid T[i] \geq m\}$$

Condiția

$$\#\{i \in \{1, \ldots, n\} \mid T[i] < m\} < \lceil n/2 \rceil$$

este deci echivalentă cu condiția

$$\#\{i \in \{1, \ldots, n\} \mid T[i] \geq m\} > n - \lceil n/2 \rceil = \lfloor n/2 \rfloor$$

Algoritmul "naiv" pentru determinarea medianei lui T constă în a sorta crescător tabloul și a extrage apoi elementul din poziția $\lceil n/2 \rceil$. Folosind *mergesort*, de exemplu, acest algoritm necesită un timp în $\Theta(n \log n)$. Putem găsi o metodă mai eficientă? Pentru a răspunde la această întrebare, vom considera o problemă mai generală.

Fie T un tablou de n elemente și fie k un întreg, $1 \leq k \leq n$. *Problema selecției* constă în găsirea celui de-al k-lea cel mai mic element al lui T, adică a elementul m pentru care avem:

$$\#\{i \in \{1, \ldots, n\} \mid T[i] < m\} < k$$

$$\#\{i \in \{1, \ldots, n\} \mid T[i] \leq m\} \geq k$$

Cu alte cuvinte, este al k-lea element în T, dacă tabloul este sortat în ordine crescătoare. De exemplu, mediana lui T este al $\lceil n/2 \rceil$-lea cel mai mic element al lui T. Deoarece $\lceil n/2 \rceil = \lfloor (n+1)/2 \rfloor = (n+1)$ **div** 2, mediana lui T este totodată al $((n+1)$ **div** $2)$-lea cel mai mic element al lui T.

Următorul algoritm, încă nu pe deplin specificat, rezolvă problema selecției într-un mod similar cu *quicksort* dar și cu *binsearch*.

```
function selection(T[1 .. n], k)
    {găsește al k-lea cel mai mic element al lui T;
     se presupune că 1 ≤ k ≤ n}
    if n este mic then  sortează T
                        return T[k]
    p ← un element pivot din T[1 .. n]
    u ← #{i ∈ {1, ..., n} | T[i] < p}
    v ← #{i ∈ {1, ..., n} | T[i] ≤ p}
    if u ≥ k then
        array U[1 .. u]
        U ← elementele din T mai mici decât p
        {cel de-al k-lea cel mai mic element al lui T este
         și cel de-al k-lea cel mai mic element al lui U}
        return selection (U, k)
    if v ≥ k then {am găsit!} return p
```

```
{situația când u < k și v < k}
array V[1 .. n–v]
V ← elementele din T mai mari decât p
{cel de-al k-lea cel mai mic element al lui T este
  și cel de-al (k–v)-lea cel mai mic element al lui V}
return selection(V, k–v)
```

Care element din T să fie ales ca pivot? O alegere naturală este mediana lui T, astfel încât U și V să fie de mărimi cât mai apropiate (chiar dacă cel mult unul din aceste subtablouri va fi folosit într-un apel recursiv). Dacă în algoritmul *selection* alegerea pivotului se face prin atribuirea

$$p \leftarrow selection(T, (n+1) \textbf{ div } 2)$$

ajungem însă la un cerc vicios.

Să analizăm algoritmul de mai sus, presupunând, pentru început, că găsirea medianei este o operație elementară. Din definiția medianei, rezultă că $u < \lceil n/2 \rceil$ și $v \geq \lceil n/2 \rceil$. Obținem atunci relația $n-v \leq \lfloor n/2 \rfloor$. Dacă există un apel recursiv, atunci tablourile U și V conțin fiecare cel mult $\lfloor n/2 \rfloor$ elemente. Restul operațiilor necesită un timp în ordinul lui n. Fie $t_m(n)$ timpul necesar acestei metode, în cazul cel mai nefavorabil, pentru a găsi al k-lea cel mai mic element al unui tablou de n elemente. Avem

$$t_m(n) \in O(n) + \max\{t_m(i) \mid i \leq \lfloor n/2 \rfloor\}$$

De aici se deduce (Exercițiul 7.17) că $t_m \in O(n)$.

Ce facem însă dacă trebuie să ținem cont și de timpul pentru găsirea pivotului? Putem proceda ca în cazul *quicksort*-ului și să renunțăm la mediană, alegând ca pivot primul element al tabloului. Algoritmul *selection* astfel precizat are timpul pentru cazul mediu în ordinul exact al lui n. Pentru cazul cel mai nefavorabil, se obține însă un timp în ordinul lui n^2.

Putem evita acest caz cel mai nefavorabil cu timp pătratic, fără să sacrificăm comportarea liniară pentru cazul mediu. Ideea este să găsim rapid o aproximare bună pentru mediană. Presupunând $n \geq 5$, vom determina pivotul prin atribuirea

$$p \leftarrow pseudomed(T)$$

unde algoritmul *pseudomed* este:

```
function pseudomed(T[1 .. n])
    {găsește o aproximare a medianei lui T}
    s ← n div 5
    array S[1 .. s]
    for i ← 1 to s do S[i] ← adhocmed5(T[5i–4 .. 5i])
    return selection(S, (s+1) div 2)
```

Algoritmul *adhocmed*5 este elaborat special pentru a găsi mediana a exact cinci elemente. Să notăm că *adhocmed*5 necesită un timp în $O(1)$.

Fie m aproximarea medianei tabloului T, găsită prin algoritmul *pseudomed*. Deoarece m este mediana tabloului S, avem

$$\#\{i \in \{1, \ldots, s\} \mid S[i] \leq m\} \geq \lceil s/2 \rceil$$

Fiecare element din S este mediana a cinci elemente din T. În consecință, pentru fiecare i, astfel încât $S[i] \leq m$, există i_1, i_2, i_3 între $5i-4$ și $5i$, astfel ca

$$T[i_1] \leq T[i_2] \leq T[i_3] = S[i] \leq m$$

Deci,

$$\#\{i \in \{1, \ldots, n\} \mid T[i] \leq m\} \geq 3\lceil s/2 \rceil = 3\lceil \lfloor n/5 \rfloor /2 \rceil$$
$$= 3\lceil \lceil (n-4)/5 \rceil /2 \rceil = 3\lceil (n-4)/10 \rceil \geq (3n-12)/10$$

Similar, din relația

$$\#\{i \in \{1, \ldots, s\} \mid S[i] < m\} < \lceil s/2 \rceil$$

care este echivalentă cu

$$\#\{i \in \{1, \ldots, s\} \mid S[i] \geq m\} > \lfloor s/2 \rfloor$$

deducem

$$\#\{i \in \{1, \ldots, n\} \mid T[i] \geq m\} > 3\lfloor \lfloor n/5 \rfloor /2 \rfloor$$
$$= 3\lfloor n/10 \rfloor = 3\lceil (n-9)/10 \rceil \geq (3n-27)/10$$

Deci,

$$\#\{i \in \{1, \ldots, n\} \mid T[i] < m\} < (7n+27)/10$$

În concluzie, m aproximează mediana lui T, fiind al k-lea cel mai mic element al lui T, unde k este aproximativ între $3n/10$ și $7n/10$. O interpretare grafică ne va ajuta să înțelegem mai bine aceste relații. Să ne imaginăm elementele lui T dispuse pe cinci linii, cu posibila excepție a cel mult patru elemente (Figura 7.1). Presupunem că fiecare din cele $\lfloor n/5 \rfloor$ coloane este ordonată nedescrescător, de sus în jos. De asemenea, presupunem că linia din mijloc (corespunzătoare tabloului S din algoritm) este ordonată nedescrescător, de la stânga la dreapta. Elementul subliniat corespunde atunci medianei lui S, deci lui m. Elementele din interiorul dreptunghiului sunt mai mici sau egale cu m. Dreptunghiul conține aproximativ 3/5 din jumătatea elementelor lui T, deci în jur de $3n/10$ elemente.

Presupunând că folosim "$p \leftarrow pseudomed(T)$", adică pivotul este pseudomediana, fie $t(n)$ timpul necesar algoritmului *selection*, în cazul cel mai nefavorabil, pentru a găsi al k-lea cel mai mic element al unui tablou de n elemente. Din inegalitățile

$$\#\{i \in \{1, \ldots, n\} \mid T[i] \leq m\} \geq (3n-12)/10$$

$$\#\{i \in \{1, \ldots, n\} \mid T[i] < m\} < (7n+27)/10$$

rezultă că, pentru n suficient de mare, tablourile U și V au cel mult $3n/4$ elemente fiecare. Deducem relația

$$t(n) \in O(n) + t(\lfloor n/5 \rfloor) + \max\{t(i) \mid i \leq 3n/4\} \qquad (*)$$

Vom arăta că $t \in \Theta(n)$. Să considerăm funcția $f : \mathbf{N} \to \mathbf{R}^*$, definită prin recurența

$$f(n) = f(\lfloor n/5 \rfloor) + f(\lfloor 3n/4 \rfloor) + n$$

pentru $n \in \mathbf{N}$. Prin inducție constructivă, putem demonstra că există constanta reală pozitivă a astfel încât $f(n) \leq an$ pentru orice $n \in \mathbf{N}$. Deci, $f \in O(n)$. Pe de altă parte, există constanta reală pozitivă c, astfel încât $t(n) \leq cf(n)$ pentru orice $n \in \mathbf{N}^+$. Este adevărată atunci și relația $t \in O(n)$. Deoarece orice algoritm care rezolvă problema selecției are timpul de execuție în $\Omega(n)$, rezultă $t \in \Omega(n)$, deci, $t \in \Theta(n)$.

Generalizând, vom încerca să aproximăm mediana nu numai prin împărțire la cinci, ci prin împărțire la un întreg q oarecare, $1 < q \leq n$. Din nou, pentru n suficient de mare, tablourile U și V au cel mult $3n/4$ elemente fiecare. Relația (*) devine

$$t(n) \in O(n) + t(\lfloor n/q \rfloor) + \max\{t(i) \mid i \leq 3n/4\} \qquad (**)$$

Dacă $1/q + 3/4 < 1$, adică dacă numărul de elemente asupra cărora operează cele două apeluri recursive din (**) este în scădere, deducem, într-un mod similar cu situația când $q = 5$, că timpul este tot liniar. Deoarece pentru orice $q \geq 5$ inegalitatea precedentă este verificată, rămâne deschisă problema alegerii unui q pentru care să obținem o constantă multiplicativă cât mai mică.

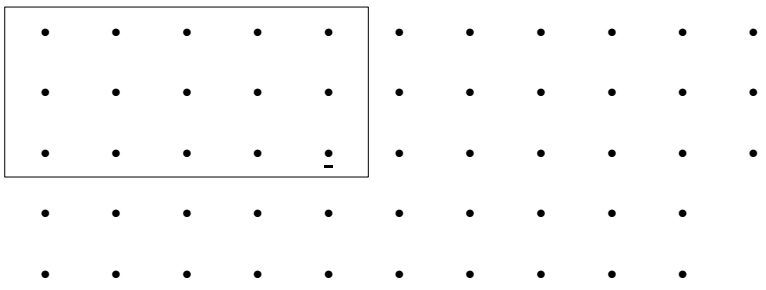


Figura 7.1 Vizualizarea pseudomedianei.

În particular, putem determina mediana unui tablou în timp liniar, atât pentru cazul mediu cât și pentru cazul cel mai nefavorabil. Față de algoritmul "naiv", al cărui timp este în ordinul lui $n \log n$, îmbunătățirea este substanțială.

7.7 O problemă de criptologie

Alice și Bob doresc să comunice anumite secrete prin telefon. Convorbirea telefonică poate fi însă ascultată și de Eva. În prealabil, Alice și Bob nu au stabilit nici un protocol de codificare și pot face acum acest lucru doar prin telefon. Eva va asculta însă și ea modul de codificare. Problema este cum să comunice Alice și Bob, astfel încât Eva să nu poată descifra codul, cu toate că va cunoaște și ea protocolul de codificare[*].

Pentru început, Alice și Bob convin în mod deschis asupra unui întreg p cu câteva sute de cifre și asupra unui alt întreg g între 2 și $p-1$. Securitatea secretului nu este compromisă prin faptul că Eva află aceste numere.

La pasul doi, Alice și Bob aleg la întîmplare câte un întreg A, respectiv B, mai mici decât p, fără să-și comunice aceste numere. Apoi, Alice calculează $a = g^A \bmod p$ și transmite rezultatul lui Bob; similar, Bob transmite lui Alice valoarea $b = g^B \bmod p$. În final, Alice calculează $x = b^A \bmod p$, iar Bob calculează $y = a^B \bmod p$. Vor ajunge la același rezultat, deoarece $x = y = g^{AB} \bmod p$. Această valoare este deci cunoscută de Alice și Bob, dar rămâne necunoscută lui Eva. Evident, nici Alice și nici Bob nu pot controla direct care va fi această valoare. Deci ei nu pot folosi acest protocol pentru a schimba în mod direct un anumit mesaj. Valoarea rezultată poate fi însă cheia unui sistem criptografic convențional.

Interceptând convorbirea telefonică, Eva va putea cunoaște în final următoarele numere: p, q, a și b. Pentru a-l deduce pe x, ea trebuie să găsească un întreg A', astfel încât $a = g^{A'} \bmod p$ și să procedeze apoi ca Alice pentru a-l calcula pe $x' = b^{A'} \bmod p$. Se poate arăta (Exercițiul 7.21) că $x' = x$, deci că Eva poate calcula astfel corect secretul lui Alice și Bob.

Calcularea lui A' din p, g și a este cunoscută ca *problema logaritmului discret* și poate fi realizată de următorul algoritm.

---

[*] O primă soluție a acestei probleme a fost dată în 1976 de W. Diffie și M. E. Hellman. Între timp s-au mai propus și alte protocoale.

```
function dlog(g, a, p)
    A ← 0; k ← 1
    repeat
        A ← A+1
        k ← kg
    until (a = k mod p) or (A = p)
    return A
```

Dacă logaritmul nu există, funcția *dlog* va returna valoarea *p*. De exemplu, nu există un întreg *A*, astfel încât $3 = 2^A$ **mod** 7. Algoritmul de mai sus este însă extrem de ineficient. Dacă *p* este un număr prim impar, atunci este nevoie în medie de *p*/2 repetări ale buclei **repeat** pentru a ajunge la soluție (presupunând că aceasta există). Dacă pentru efecuarea unei bucle este necesară o microsecundă, atunci timpul de execuție al algoritmului poate fi mai mare decât vârsta Pământului! Iar aceasta se întâmplă chiar și pentru un număr zecimal *p* cu doar 24 de cifre.

Cu toate că există și algoritmi mai rapizi pentru calcularea logaritmilor discreți, nici unul nu este suficient de eficient dacă *p* este un număr prim cu câteva sute de cifre. Pe de altă parte, nu se cunoaște până în prezent un alt mod de a-l obține pe *x* din *p*, *g*, *a* și *b*, decât prin calcularea logaritmului discret.

Desigur, Alice și Bob trebuie să poată calcula rapid exponențierile de forma $a = g^A$ **mod** *p*, căci altfel ar fi și ei puși în situația Evei. Următorul algoritm pentru calcularea exponențierii nu este cu nimic mai subtil sau eficient decât cel pentru logaritmul discret.

```
function dexpo1(g, A, p)
    a ← 1
    for i ← 1 to A do a ← ag
    return a mod p
```

Faptul că *x y z* **mod** *p* = ((*x y* **mod** *p*) *z*) **mod** *p* pentru orice *x*, *y*, *z* și *p*, ne permite să evităm memorarea unor numere extrem de mari. Obținem astfel o primă îmbunătățire:

```
function dexpo2(g, A, p)
    a ← 1
    for i ← 1 to A do a ← ag mod p
    return a
```

Din fericire pentru Alice și Bob, există un algoritm eficient pentru calcularea exponențierii și care folosește reprezentarea binară a lui *A*. Să considerăm pentru început următorul exemplu

$$x^{25} = (((x^2x)^2)^2)^2x$$

L-am obținut deci pe x^{25} prin doar două înmulțiri și patru ridicări la pătrat. Dacă în expresia

$$x^{25} = (((x^2x)^2 1)^2 1)^2 x$$

înlocuim fiecare x cu un 1 și fiecare 1 cu un 0, obținem secvența 11001, adică reprezentarea binară a lui 25. Formula precedentă pentru x^{25} are această formă, deoarece $x^{25} = x^{24}x$, $x^{24} = (x^{12})^2$ etc. Rezultă un algoritm divide et impera în care se testează în mod recursiv dacă exponentul curent este par sau impar.

```
function dexpo(g, A, p)
    if A = 0 then return 1
    if A este impar  then  a ← dexpo(g, A−1, p)
                           return (ag mod p)
                     else  a ← dexpo(g, A/2, p)
                           return (aa mod p)
```

Fie $h(A)$ numărul de înmulțiri modulo p efectuate atunci când se calculează *dexpo*(g, A, p), inclusiv ridicarea la pătrat. Atunci,

$$h(A) = \begin{cases} 0 & \text{pentru } A = 0 \\ 1 + h(A-1) & \text{pentru } A \text{ impar} \\ 1 + h(A/2) & \text{altfel} \end{cases}$$

Dacă $M(p)$ este limita superioară a timpului necesar înmulțirii modulo p a două numere naturale mai mici decât p, atunci calcularea lui *dexpo*(g, A, p) necesită un timp în $O(M(p)\, h(A))$. Mai mult, se poate demonstra că timpul este în $O(M(p) \log A)$, ceea ce este rezonabil. Ca și în cazul căutării binare, algoritmul *dexpo* este mai curând un exemplu de simplificare decât de tehnică divide et impera.

Vom înțelege mai bine acest algoritm, dacă considerăm și o versiune iterativă a lui.

```
function dexpoiter1(g, A, p)
    c ← 0; a ← 1
    {fie A_k A_{k−1} ... A_0 reprezentarea binară a lui A}
    for i ← k downto 0 do
        c ← 2c
        a ← aa mod p
        if A_i = 1 then  c ← c + 1
                         a ← ag mod p
    return a
```

Fiecare iterație folosește una din identitățile

$$g^{2c} \textbf{ mod } p = (g^c)^2 \textbf{ mod } p$$

$$g^{2c+1} \textbf{ mod } p = g(g^c)^2 \textbf{ mod } p$$

în funcție de valoarea lui A_i (dacă este 0, respectiv 1). La sfârșitul pasului i, valoarea lui c, în reprezentare binară, este $\overline{A_k A_{k-1} \ldots A_i}$. Reprezentrea binară a lui A este parcursă de la stânga spre dreapta, invers ca la algoritmul *dexpo*. Variabila c a fost introdusă doar pentru a înțelege mai bine cum funcționează algoritmul și putem, desigur, să o eliminăm.

Dacă parcurgem reprezentarea binară a lui A de la dreapta spre stânga, obținem un alt algoritm iterativ la fel de interesant.

```
function dexpoiter2(g, A, p)
    n ← A; y ← g; a ← 1
    while n > 0 do
        if n este impar then a ← ay mod p
        y ← yy mod p
        n ← n div 2
    return a
```

Pentru a compara aceşti trei algoritmi, vom considera următorul exemplu. Algoritmul *dexpo* îl calculează pe x^{15} sub forma $(((1\,x)^2x)^2x)^2x$, cu șapte înmulțiri; algoritmul *dexpoiter*1 sub forma $(((1^2x)^2x)^2x)^2x$, cu opt înmulțiri; iar *dexpoiter*2 sub forma $1\,x\,x^2x^4x^8$, tot cu opt înmulțiri (ultima din acestea fiind pentru calcularea inutilă a lui x^{16}).

Se poate observa că nici unul din aceşti algoritmi nu minimizează numărul de înmulțiri efectuate. De exemplu, x^{15} poate fi obținut prin șase înmulțiri, sub forma $((x^2x)^2x)^2x$. Mai mult, x^{15} poate fi obținut prin doar cinci înmulțiri (Exercițiul 7.22).

7.8 Înmulțirea matricilor

Pentru matricile A și B de $n \times n$ elemente, dorim să obținem matricea produs $C = AB$. Algoritmul clasic provine direct din definiția înmulțirii a două matrici și necesită n^3 înmulțiri şi $(n-1)n^2$ adunări scalare. Timpul necesar pentru calcularea matricii C este deci în $\Theta(n^3)$. Problema pe care ne-o punem este să găsim un algoritm de înmulțire matricială al cărui timp să fie într-un ordin mai mic decât n^3. Pe de altă parte, este clar că $\Omega(n^2)$ este o limită inferioară pentru orice algoritm de înmulțire matricială, deoarece trebuie în mod necesar să parcurgem cele n^2 elemente ale lui C.

Strategia divide et impera sugerează un alt mod de calcul a matricii C. Vom presupune în continuare că n este o putere a lui doi. Partiționăm pe A și B în câte patru submatrici de $n/2 \times n/2$ elemente fiecare. Matricea produs C se poate calcula conform formulei pentru produsul matricilor de 2×2 elemente:

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

unde

$$C_{11} = A_{11}B_{11} + A_{12}B_{21} \qquad C_{12} = A_{11}B_{12} + A_{12}B_{22}$$
$$C_{21} = A_{21}B_{11} + A_{22}B_{21} \qquad C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

Pentru $n = 2$, înmulțirile și adunările din relațiile de mai sus sunt scalare; pentru $n > 2$, aceste operații sunt între matrici de $n/2 \times n/2$ elemente. Operația de adunare matricială este cea clasică. În schimb, pentru fiecare înmulțire matricială, aplicăm recursiv aceste partiționări, până când ajungem la submatrici de 2×2 elemente.

Pentru a obține matricea C, este nevoie de opt înmulțiri și patru adunări de matrici de $n/2 \times n/2$ elemente. Două matrici de $n/2 \times n/2$ elemente se pot aduna într-un timp în $\Theta(n^2)$. Timpul total pentru algoritmul divide et impera rezultat este

$$t(n) \in 8t(n/2) + \Theta(n^2)$$

Definim funcția

$$f(n) = \begin{cases} 1 & \text{pentru } n = 1 \\ 8f(n/2) + n^2 & \text{pentru } n \neq 1 \end{cases}$$

Din Proprietatea 5.2 rezultă că $f \in \Theta(n^3)$. Procedând ca în Secțiunea 5.1.2, deducem că $t \in \Theta(f) = \Theta(n^3)$, ceea ce înseamnă că nu am câștigat încă nimic față de metoda clasică.

În timp ce înmulțirea matricilor necesită un timp cubic, adunarea matricilor necesită doar un timp pătratic. Este, deci, de dorit ca în formulele pentru calcularea submatricilor C să folosim mai puține înmulțiri, chiar dacă prin aceasta mărim numărul de adunări. Este însă acest lucru și posibil? Răspunsul este afirmativ. În 1969, Strassen a descoperit o metodă de calculare a submatricilor C_{ij}, care utilizează 7 înmulțiri și 18 adunări și scăderi. Pentru început, se calculează șapte matrici de $n/2 \times n/2$ elemente:

$$\begin{aligned} P &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ Q &= (A_{21} + A_{22})B_{11} \\ R &= A_{11}(B_{12} - B_{22}) \\ S &= A_{22}(B_{21} - B_{11}) \\ T &= (A_{11} + A_{12})B_{22} \\ U &= (A_{21} - A_{11})(B_{11} + B_{22}) \\ V &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

Este uşor de verificat că matricea produs C se obţine astfel:

$$C_{11} = P + S - T + V \qquad C_{12} = R + T$$
$$C_{21} = Q + S \qquad C_{22} = P + R - Q + U$$

Timpul total pentru noul algoritm divide et impera este

$$t(n) \in 7t(n/2) + \Theta(n^2)$$

şi în mod similar deducem că $t \in \Theta(n^{\lg 7})$. Deoarece $\lg 7 < 2{,}81$, rezultă că $t \in O(n^{2,81})$. Algoritmul lui Strassen este deci mai eficient decât algoritmul clasic de înmulţire matricială.

Metoda lui Strassen nu este unică: s-a demonstrat că există exact 36 de moduri diferite de calcul a submatricilor C_{ij}, fiecare din aceste metode utilizând 7 înmulţiri.

Limita $O(n^{2,81})$ poate fi şi mai mult redusă dacă găsim un algoritm de înmulţire a matricilor de 2×2 elemente cu mai puţin de şapte înmulţiri. S-a demonstrat însă că acest lucru nu este posibil. O altă metodă este de a găsi algoritmi mai eficienţi pentru înmulţirea matricilor de dimensiuni mai mari decât 2×2 şi de a descompune recursiv până la nivelul acestor submatrici. Datorită constantelor multiplicative implicate, exceptând algoritmul lui Strassen, nici unul din aceşti algoritmi nu are o valoare practică semnificativă.

Pe calculator, s-a putut observa că, pentru $n \geq 40$, algoritmul lui Strassen este mai eficient decât metoda clasică. În schimb, algoritmul lui Strassen foloseşte memorie suplimentară.

Poate că este momentul să ne întrebăm de unde provine acest interes pentru înmulţirea matricilor. Importanţa acestor algoritmi* derivă din faptul că operaţii frecvente cu matrici (cum ar fi inversarea sau calculul determinantului) se bazează pe înmulţiri de matrici. Astfel, dacă notăm cu $f(n)$ timpul necesar pentru a înmulţi două matrici de $n \times n$ elemente şi cu $g(n)$ timpul necesar pentru a inversa o matrice nesingulară de $n \times n$ elemente, se poate arăta că $f \in \Theta(g)$.

7.9 Înmulţirea numerelor întregi mari

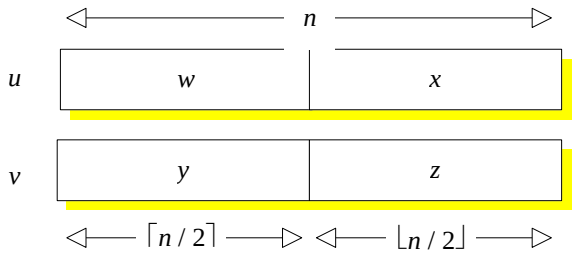
Pentru anumite aplicaţii, trebuie să considerăm numere întregi foarte mari. Dacă aţi implementat algoritmii pentru generarea numerelor lui Fibonacci, probabil că

* S-au propus şi metode complet diferite. Astfel, D. Coppersmith şi S. Winograd au găsit în 1987 un algoritm cu timpul în $O(n^{2,376})$.

v-ați confruntat deja cu această problemă. Același lucru s-a întâmplat în 1987, atunci când s-au calculat primele 134 de milioane de cifre ale lui π. În criptologie, numerele întregi mari sunt de asemenea extrem de importante (am văzut acest lucru în Secțiunea 7.7). Operațiile aritmetice cu operanzi întregi foarte mari nu mai pot fi efectuate direct prin hardware, deci nu mai putem presupune, ca până acum, că operațiile necesită un timp constant. Reprezentarea operanzilor în virgulă flotantă ar duce la aproximări nedorite. Suntem nevoiți deci să implementăm prin software operațiile aritmetice respective.

În cele ce urmează, vom da un algoritm divide et impera pentru înmulțirea întregilor foarte mari. Fie u și v doi întregi foarte mari, fiecare de n cifre zecimale (convenim să spunem că un întreg k are j cifre dacă $k < 10^j$, chiar dacă $k < 10^{j-1}$). Dacă $s = \lfloor n/2 \rfloor$, reprezentăm pe u și v astfel:

$$u = 10^s w + x, \quad v = 10^s y + z, \quad \text{unde } 0 \le x < 10^s,\ 0 \le z < 10^s$$



Întregii w și y au câte $\lceil n/2 \rceil$ cifre, iar întregii x și z au câte $\lfloor n/2 \rfloor$ cifre. Din relația

$$uv = 10^{2s} wy + 10^s (wz + xy) + xz$$

obținem următorul algoritm divide et impera pentru înmulțirea a două numere întregi mari.

function *înmulțire*(u, v)
  $n \leftarrow$ cel mai mic întreg astfel încât u și v să aibă fiecare n cifre
  if n este mic **then** calculează în mod clasic produsul uv
    return produsul uv astfel calculat
  $s \leftarrow n$ **div** 2
  $w \leftarrow u$ **div** 10^s ; $x \leftarrow u$ **mod** 10^s
  $y \leftarrow v$ **div** 10^s ; $z \leftarrow v$ **mod** 10^s
  return *înmulțire*(w, y) $\times 10^{2s}$
    + (*înmulțire*(w, z)+*înmulțire*(x, y)) $\times 10^s$
    + *înmulțire*(x, z)

Presupunând că folosim reprezentarea din Exercițiul 7.28, înmulțirile sau împărțirile cu 10^{2s} și 10^s, ca și adunările, sunt executate într-un timp liniar. Același lucru este atunci adevărat și pentru restul împărțirii întregi, deoarece

$$u \textbf{ mod } 10^s = u - 10^s w, \quad v \textbf{ mod } 10^s = v - 10^s y$$

Notăm cu $t_d(n)$ timpul necesar acestui algoritm, în cazul cel mai nefavorabil, pentru a înmulți doi întregi de n cifre. Avem

$$t_d(n) \in 3t_d(\lceil n/2 \rceil) + t_d(\lfloor n/2 \rfloor) + \Theta(n)$$

Dacă n este o putere a lui 2, această relație devine

$$t_d(n) \in 4t_d(n/2) + \Theta(n)$$

Folosind Proprietatea 5.2, obținem relația $t_d \in \Theta(n^2)$. (Se observă că am reîntâlnit un exemplu din Secțiunea 5.3.5). Înmulțirea clasică necesită însă tot un timp pătratic (Exercițiul 5.29). Nu am câștigat astfel nimic; dimpotrivă, am reușit să mărim constanta multiplicativă!

Ideea care ne va ajuta am mai folosit-o la metoda lui Strassen (Secțiunea 7.8). Deoarece înmulțirea întregilor mari este mult mai lentă decât adunarea, încercăm să reducem numărul înmulțirilor, chiar dacă prin aceasta mărim numărul adunărilor. Adică, încercăm să calculăm wy, $wz+xy$ și xz prin mai puțin de patru înmulțiri. Considerând produsul

$$r = (w+x)(y+z) = wy + (wz+xy) + xz$$

observăm că putem înlocui ultima linie din algoritm cu

$r \leftarrow$ *înmulț*$(w+x, y+z)$
$p \leftarrow$ *înmulț*(w, y); $q \leftarrow$ *înmulț*(x, z)
return $10^{2s}p + 10^s(r-p-q) + q$

Fie $t(n)$ timpul necesar algoritmului modificat pentru a înmulți doi întregi, fiecare cu *cel mult* n cifre. Ținând cont că $w+x$ și $y+z$ pot avea cel mult $1+\lceil n/2 \rceil$ cifre, obținem

$$t(n) \in t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + t(1+\lceil n/2 \rceil) + O(n)$$

Prin definiție, funcția t este nedescrescătoare. Deci,

$$t(n) \in 3t(1+\lceil n/2 \rceil) + O(n)$$

Notând $T(n) = t(n+2)$ și presupunând că n este o putere a lui 2, obținem

$$T(n) \in 3T(n/2) + O(n)$$

Prin metoda iterației (ca în Exercițiul 7.24), puteți arăta că

$$T \in O(n^{\lg 3} \mid n \text{ este o putere a lui 2})$$

Sau, mai elegant, puteți ajunge la același rezultat aplicând o schimbare de variabilă (o recurență asemănătoare a fost discutată în Secțiunea 5.3.5). Deci,

$$t \in O(n^{\lg 3} \mid n \text{ este o putere a lui 2})$$

Ținând din nou cont că t este nedescrescătoare, aplicăm Proprietatea 5.1 și obținem $t \in O(n^{\lg 3})$.

În concluzie, este posibil să înmulțim doi întregi de n cifre într-un timp în $O(n^{\lg 3})$, deci și în $O(n^{1,59})$. Ca și la metoda lui Strassen, datorită constantelor multiplicative implicate, acest algoritm este interesant în practică doar pentru valori mari ale lui n. O implementare bună nu va folosi probabil baza 10, ci baza cea mai mare pentru care hardware-ul permite ca două "cifre" să fie înmulțite direct.

7.10 Exerciții

7.1 Demonstrați că procedura *binsearch* se termină într-un număr finit de pași (nu ciclează).

Indicație: Arătați că *binrec*($T[i \,..\, j]$, x) este apelată întotdeauna cu $i \leq j$ și că *binrec*($T[i \,..\, j]$, x) apelează *binrec*($T[u \,..\, v]$, x) întotdeauna astfel încât

$$v-u < j-i$$

7.2 Se poate înlocui în algoritmul *iterbin*1:

i) "$k \leftarrow (i+j+1)$ **div** 2" cu "$k \leftarrow (i+j)$ **div** 2"?

ii) "$i \leftarrow k$" cu "$i \leftarrow k+1$"?

iii) "$j \leftarrow k-1$" cu "$j \leftarrow k$"?

7.3 Observați că bucla **while** din algoritmul *insert* (Secțiunea 1.3) folosește o căutare secvențială (de la coadă la cap). Să înlocuim această căutare secvențială cu o căutare binară. Pentru cazul cel mai nefavorabil, ajungem oare acum ca timpul pentru sortarea prin inserție să fie în ordinul lui $n \log n$?

7.4 Arătați că timpul pentru *iterbin*2 este în $\Theta(1)$, $\Theta(\log n)$, $\Theta(\log n)$ pentru cazurile cel mai favorabil, mediu și respectiv, cel mai nefavorabil.

7.5 Fie $T[1 \,..\, n]$ un tablou ordonat crescător de întregi diferiți, unii putând fi negativi. Dați un algoritm cu timpul în $O(\log n)$ pentru cazul cel mai nefavorabil, care găsește un index i, $1 \leq i \leq n$, cu $T[i] = i$, presupunând că acest index există.

7.6 *Rădăcina pătrată întreagă* a lui $n \in \mathbf{N}$ este prin definiție acel $p \in \mathbf{N}$ pentru care $p \leq \sqrt{n} < p+1$. Presupunând că nu avem o funcție radical, elaborați un algoritm care îl găsește pe p într-un timp în $O(\log n)$.

Soluție: Se apelează *pătrat*(0, *n*+1, *n*), *pătrat* fiind funcția

```
function pătrat(a, b, n)
    if a = b−1 then return a
    m ← (a+b) div 2
    if m^2 ≤ n   then  pătrat(m, b, n)
                 else  pătrat(a, m, n)
```

7.7 Fie tablourile $U[1 .. N]$ și $V[1 .. M]$, ordonate crescător. Elaborați un algoritm cu timpul de execuție în $\Theta(N+M)$, care să interclaseze cele două tablouri. Rezultatul va fi trecut în tabloul $T[1 .. N+M]$.

Soluție: Iată o primă variantă a acestui algoritm:

```
i, j, k ← 1
while i ≤ N and j ≤ M do
    if U[i] ≤ V[j]   then  T[k] ← U[i]
                           i ← i+1
                     else  T[k] ← V[j]
                           j ← j+1
    k ← k+1
if i > N   then for h ← j to M do
                T[k] ← V[h]
                k ← k+1
           else for h ← i to N do
                T[k] ← U[h]
                k ← k+1
```

Se poate obține un algoritm și mai simplu, dacă se presupune că avem acces la locațiile $U[N+1]$ și $V[M+1]$, pe care le vom inițializa cu o valoare maximală și le vom folosi ca "santinele":

```
i, j ← 1
U[N+1], V[M+1] ← +∞
for k ← 1 to N+M do
    if U[i] < V[j]   then  T[k] ← U[i]
                           i ← i+1
                     else  T[k] ← V[j]
                           j ← j+1
```

Mai rămâne să analizați eficiența celor doi algoritmi.

7.8 Modificați algoritmul *mergesort* astfel încât T să fie separat nu în două, ci în trei părți de mărimi cât mai apropiate. Analizați algoritmul obținut.

7.9 Arătați că, dacă în algoritmul *mergesort* separăm pe T în tabloul U, având $n-1$ elemente, și tabloul V, având un singur element, obținem un algoritm de sortare cu timpul de execuție în $\Theta(n^2)$. Acest nou algoritm seamănă cu unul dintre algoritmii deja cunoscuți. Cu care anume?

7.10 Iată și o altă procedură de pivotare:

```
procedure pivot1(T[i .. j], l)
    p ← T[i]
    l ← i
    for k ← i+1 to j do
        if T[k] ≤ p  then   l ← l+1
                            interschimbă T[k] și T[l]
    interschimbă T[i] și T[l]
```

Argumentați de ce procedura este corectă și analizați eficiența ei. Comparați numărul maxim de interschimbări din procedurile *pivot* și *pivot*1. Este oare rentabil ca în algoritmul *quicksort* să înlocuim procedura *pivot* cu procedura *pivot*1?

7.11 Argumentați de ce un apel *funny-sort*(T[1 ..n]) al următorului algoritm sortează corect elementele tabloului T[1 .. n].

```
procedure funny-sort(T[i .. j])
    if T[i] > T[ j] then interschimbă T[i] și T[ j]
    if i < j−1 then  k ← ( j−i+1) div 3
                     funny-sort(T[i .. j−k])
                     funny-sort(T[i+k .. j])
                     funny-sort(T[i .. j−k])
```

Este oare acest simpatic algoritm și eficient?

7.12 Este un lucru elementar să găsim un algoritm care determină minimul dintre elementele unui tablou T[1 .. n] și utilizează pentru aceasta $n-1$ comparații între elemente ale tabloului. Mai mult, orice algoritm care determină prin comparații minimul elementelor din T efectuează în mod necesar cel puțin $n-1$ comparații. În anumite aplicații, este nevoie să găsim atât minimul cât și maximul

dintr-o mulțime de n elemente. Iată un algoritm care determină minimul și maximul dintre elementele tabloului $T[1 .. n]$:

```
procedure fmaxmin1(T[1 .. n], max, min)
    max, min ← T[1]
    for i ← 2 to n do
        if max < T[i]  then max ← T[i]
        if min > T[i]  then min ← T[i]
```

Acest algoritm efectuează $2(n-1)$ comparații între elemente ale lui T. Folosind tehnica divide et impera, elaborați un algoritm care să determine minimul și maximul dintre elementele lui T prin mai puțin de $2(n-1)$ comparații. Puteți presupune că n este o putere a lui 2.

Soluție: Un apel *fmaxmin*2($T[1 .. n]$, *max*, *min*) al următorului algoritm găsește minimul și maximul cerute

```
procedure fmaxmin2(T[i .. j], max, min)
    case  i = j : max, min ← T[i]
          i = j–1 : if T[i] < T[ j]   then  max ← T[ j]
                                            min ← T[i]
                                      else  max ← T[i]
                                            min ← T[ j]
          otherwise :  m ← (i+j) div 2
                       fmaxmin2(T[i .. m], smax, smin)
                       fmaxmin2(T[m+1 .. j], dmax, dmin)
                       max ← maxim(smax, dmax)
                       min ← minim(smin, dmin)
```

Funcțiile *maxim* și *minim* determină, prin câte o singură comparație, maximul, respectiv minimul, a două elemente.

Putem deduce că atât *fmaxmin*1, cât și *fmaxmin*2 necesită un timp în $\Theta(n)$ pentru a găsi minimul și maximul într-un tablou de n elemente. Constanta multiplicativă asociată timpului în cele două cazuri diferă însă. Notând cu $C(n)$ numărul de comparații între elemente ale tabloului T efectuate de procedura *fmaxmin*2, obținem recurența

$$C(n) = \begin{cases} 0 & \text{pentru } n = 1 \\ 1 & \text{pentru } n = 2 \\ C(\lfloor n/2 \rfloor) + C(\lceil n/2 \rceil) + 2 & \text{pentru } n > 2 \end{cases}$$

Considerăm $n = 2^k$ și folosim metoda iterației:

$$C(n) = 2C(n/2) + 2 = \ldots = 2^{k-1}C(2) + \sum_{i=1}^{k-1} 2^i = 2^{k-1} + 2^k - 2 = 3n/2 - 2$$

Algoritmul *fmaxmin*2 necesită cu 25% mai puține comparații decât *fmaxmin*1. Se poate arăta că nici un algoritm bazat pe comparații nu poate folosi mai puțin de $3n/2-2$ comparații. În acest sens, *fmaxmin*2 este, deci, optim.

Este procedura *fmaxmin*2 mai eficientă și în practică? Nu în mod necesar. Analiza ar trebui să considere și numărul de comparații asupra indicilor de tablou, precum și timpul necesar pentru rezolvarea apelurilor recursive în *fmaxmin*2. De asemenea, ar trebui să cunoaștem și cu cât este mai costisitoare o comparație de elemente ale lui *T*, decât o comparație de indici (adică, de întregi).

7.13 În ce constă similaritatea algoritmului *selection* cu algoritmul *i) quicksort* și *ii) binsearch*?

7.14 Generalizați procedura *pivot*, partiționând tabloul *T* în trei secțiuni $T[1 .. i-1]$, $T[i .. j]$, $T[j+1 .. n]$, conținând elementele lui *T* mai mici decât *p*, egale cu *p* și respectiv, mai mari decât *p*. Valorile *i* și *j* vor fi calculate în procedura de pivotare și vor fi returnate prin această procedură.

7.15 Folosind ca model versiunea iterativă a căutării binare și rezultatul Exercițiului 7.14, elaborați un algoritm nerecursiv pentru problema selecției.

7.16 Analizați următoarea variantă a algoritmului *quicksort*.

```
procedure quicksort-modificat(T[1 .. n])
    if n = 2 and T[2] < T[1]
        then interschimbă T[1] și T[2]
        else if n > 2 then
                p ← selection(T, (n+1) div 2)
                arrays U[1 .. (n+1) div 2 ], V[1 .. n div 2]
                U ←  elementele din T mai mici decât p
                     și, în completare, elemente egale cu p
                V ←  elementele din T mai mari decât p
                     și, în completare, elemente egale cu p
                quicksort-modificat(U)
                quicksort-modificat(V)
```

7.17 Dacă presupunem că găsirea medianei este o operație elementară, am văzut că timpul pentru *selection*, în cazul cel mai nefavorabil, este

$$t_m(n) \in O(n) + \max\{t_m(i) \mid i \leq \lfloor n/2 \rfloor\}$$

Demonstrați că $t_m \in O(n)$.

Soluție: Fie n_0 și d două constante astfel încât pentru $n > n_0$ avem

$$t_m(n) \le dn + \max\{t_m(i) \mid i \le \lfloor n/2 \rfloor\}$$

Putem considera că există constanta reală pozitivă c astfel încât $t_m(i) \le ci+c$, pentru $0 \le i \le n_0$. Prin ipoteza inducției specificate parțial presupunem că $t(i) \le ci+c$, pentru orice $0 \le i < n$. Atunci

$$t_m(n) \le dn+c+c\lfloor n/2 \rfloor = cn+c+dn-c\lceil n/2 \rceil \le cn+c$$

deoarece putem să alegem constanta c suficient de mare, astfel încât $c\lceil n/2 \rceil \ge dn$. Am arătat deci prin inducție că, dacă c este suficient de mare, atunci $t_m(n) \le cn+c$, pentru orice $n \ge 0$. Adică, $t_m \in O(n)$.

7.18 Arătați că luând "$p \leftarrow T[1]$" în algoritmul *selection* și considerând cazul cel mai nefavorabil, determinarea celui de-al k-lea cel mai mic element al lui $T[1 .. n]$ necesită un timp de execuție în $O(n^2)$.

7.19 Fie $U[1 .. n]$ și $V[1 .. n]$ două tablouri de elemente ordonate nedescrescător. Elaborați un algoritm care să găsească mediana celor $2n$ elemente într-un timp de execuție în $O(\log n)$.

7.20 Un element x este *majoritar* în tabloul $T[1 .. n]$, dacă $\#\{i \mid T[i] = x\} > \lfloor n/2 \rfloor$. Elaborați un algoritm liniar care să determine elementul majoritar în T (dacă un astfel de element există).

7.21 Să presupunem că Eva a găsit un A' pentru care

$$a = g^{A'} \textbf{ mod } p = g^A \textbf{ mod } p$$

și că există un B, astfel încât $b = g^B \textbf{ mod } p$. Arătați că

$$x' = b^{A'} \textbf{ mod } p = b^A \textbf{ mod } p = x$$

chiar dacă $A' \ne A$.

7.22 Arătați cum poate fi calculat x^{15} prin doar cinci înmulțiri (inclusiv ridicări la pătrat).

Soluție: $x^{15} = (((x^2)^2)^2)^2 x^{-1}$

7.23 Găsiți un algoritm divide et impera pentru a calcula un termen oarecare din șirul lui Fibonacci. Folosiți proprietatea din Exercițiul 1.7. Vă ajută aceasta la înțelegerea algoritmului *fib*3 din Secțiunea 1.6.4?

Indicație: Din Exercițiul 1.7, deducem că $f_n = m_{22}^{(n-1)}$, unde $m_{22}^{(n-1)}$ este elementul de pe ultima linie și ultima coloană ale matricii M^{n-1}. Rămâne să elaborați un algoritm similar cu *dexpo* pentru a afla matricea putere M^{n-1}. Dacă, în loc de *dexpo*, folosiți ca model algoritmul *dexpoiter*2, obțineți algoritmul *fib3*.

7.24 Demonstrați că algoritmul lui Strassen necesită un timp în $O(n^{\lg 7})$, folosind de această dată metoda iterației.

Soluție: Fie două constante pozitive a și c, astfel încât timpul pentru algoritmul lui Strassen este

$$t(n) \leq 7t(n/2) + cn^2$$

pentru $n > 2$, iar $t(n) \leq a$ pentru $n \leq 2$. Obținem

$$\begin{aligned} t(n) &\leq cn^2(1+7/4+(7/4)^2+\ldots+(7/4)^{k-2}) + a7^{k-1} \\ &\leq cn^2(7/4)^{\lg n} + a7^{\lg n} \\ &= cn^{\lg 4+\lg 7-\lg 4} + an^{\lg 7} \in O(n^{\lg 7}) \end{aligned}$$

7.25 Cum ați modifica algoritmul lui Strassen pentru a înmulți matrici de $n \times n$ elemente, unde n nu este o putere a lui doi? Arătați că timpul algoritmului rezultat este tot în $\Theta(n^{\lg 7})$.

Indicație: Îl majorăm pe n până la cea mai mică putere a lui 2, completând corespunzător matricile A și B cu elemente nule.

7.26 Să presupunem că avem o primitivă grafică *box*(x, y, r), care desenează un pătrat $2r \times 2r$ centrat în (x, y), ștergând zona din interior. Care este desenul realizat prin apelul *star*(a, b, c), unde *star* este algoritmul

```
procedure star(x, y, r)
    if r > 0 then  star(x−r, y+r, r div 2)
                   star(x+r, y+r, r div 2)
                   star(x−r, y−r, r div 2)
                   star(x+r, y−r, r div 2)
                   box(x, y, r)
```

Care este rezultatul, dacă *box*(x, y, r) apare înaintea celor patru apeluri recursive? Arătați că timpul de execuție pentru un apel *star*(a, b, c) este în $\Theta(c^2)$.

7.27 Demonstrați că pentru orice întregi m și n sunt adevărate următoarele proprietăți:

i) dacă m și n sunt pare, atunci cmmdc(m, n) = 2cmmdc(m/2, n/2)

ii) dacă m este impar și n este par, atunci cmmdc(m, n) = cmmdc(m, n/2)

iii) dacă m și n sunt impare, atunci cmmdc(m, n) = cmmdc((m−n)/2, n)

Pe majoritatea calculatoarelor, operațiile de scădere, testare a parității unui întreg și împărțire la doi sunt mai rapide decât calcularea restului împărțirii întregi. Elaborați un algoritm divide et impera pentru a calcula cel mai mare divizor comun a doi întregi, evitând calcularea restului împărțirii întregi. Folosiți proprietățile de mai sus.

7.28 Găsiți o structură de date adecvată, pentru a reprezenta numere întregi mari pe calculator. Pentru un întreg cu n cifre zecimale, numărul de biți folosiți trebuie să fie în ordinul lui n. Înmulțirea și împărțirea cu o putere pozitivă a lui 10 (sau altă bază, dacă preferați) trebuie să poată fi efectuate într-un timp liniar. Adunarea și scăderea a două numere de n, respectiv m cifre trebuie să poată fi efectuate într-un timp în $\Theta(n+m)$. Permiteți numerelor să fie și negative.

7.29 Fie u și v doi întregi mari cu n, respectiv m cifre. Presupunând că folosiți structura de date din Exercițiul 7.28, arătați că algoritmul de înmulțire clasică (și cel “a la russe”) a lui u cu v necesită un timp în $\Theta(nm)$.

8. Algoritmi de programare dinamică

8.1 Trei principii fundamentale ale programării dinamice

Programarea dinamică, ca și metoda divide et impera, rezolvă problemele combinând soluțiile subproblemelor. După cum am văzut, algoritmii divide et impera partiționează problemele în subprobleme independente, rezolvă subproblemele în mod recursiv, iar apoi combină soluțiile lor pentru a rezolva problema inițială. Dacă subproblemele conțin subsubprobleme comune, în locul metodei divide et impera este mai avantajos de aplicat tehnica programării dinamice.

Să analizăm însă pentru început ce se întâmplă cu un algoritm divide et impera în această din urmă situație. Descompunerea recursivă a cazurilor în subcazuri ale aceleiași probleme, care sunt apoi rezolvate în mod independent, poate duce uneori la calcularea de mai multe ori a aceluiași subcaz, și deci, la o eficiență scăzută a algoritmului. Să ne amintim, de exemplu, de algoritmul *fib*1 din Capitolul 1. Sau, să calculăm coeficientul binomial

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & \text{pentru } 0 < k < n \\ 1 & \text{altfel} \end{cases}$$

în mod direct:

```
function C(n, k)
    if k = 0 or k = n   then   return 1
                        else   return C(n−1, k−1) + C(n−1, k)
```

Multe din valorile $C(i, j)$, $i < n$, $j < k$, sunt calculate în mod repetat (vezi Exercițiul 2.5). Deoarece rezultatul final este obținut prin adunarea a $\binom{n}{k}$ de 1, rezultă că timpul de execuție pentru un apel $C(n, k)$ este în $\Omega(\binom{n}{k})$.

Dacă memorăm rezultatele intermediare într-un tablou de forma

	0	1	2	...	$k-1$	k
0	1					
1	1	1				
2	1	2	1			
⋮						
$n-1$					$\binom{n-1}{k-1}$	$\binom{n-1}{k}$
n						$\binom{n}{k}$

(acesta este desigur triunghiul lui Pascal), obținem un algoritm mai eficient. De fapt, este suficient să memorăm un vector de lungime k, reprezentând linia curentă din triunghiul lui Pascal, pe care să-l reactualizăm de la dreapta la stânga. Noul algoritm necesită un timp în $O(nk)$. Pe această idee se bazează și algoritmul *fib*2 (Capitolul 1). Am ajuns astfel la *primul principiu* de bază al programării dinamice: evitarea calculării de mai multe ori a aceluiași subcaz, prin memorarea rezultatelor intermediare.

Putem spune că metoda divide et impera operează *de sus în jos* (*top-down*), descompunând un caz în subcazuri din ce în ce mai mici, pe care le rezolvă apoi separat. Al *doilea principiu* fundamental al programării dinamice este faptul că ea operează *de jos în sus* (*bottom-up*). Se pornește de obicei de la cele mai mici subcazuri. Combinând soluțiile lor, se obțin soluții pentru subcazuri din ce în ce mai mari, pînă se ajunge, în final, la soluția cazului inițial.

Programarea dinamică este folosită de obicei în probleme de optimizare. În acest context, conform celui de-al *treilea principiu* fundamental, programarea dinamică este utilizată pentru a optimiza o problemă care satisface *principiul optimalității*: într-o secvență optimă de decizii sau alegeri, fiecare subsecvență trebuie să fie de asemenea optimă. Cu toate că pare evident, acest principiu nu este întotdeauna valabil și aceasta se întâmplă atunci când subsecvențele nu sunt independente, adică atunci când optimizarea unei secvențe intră în conflict cu optimizarea celorlalte subsecvențe.

Pe lângă programarea dinamică, o posibilă metodă de rezolvare a unei probleme care satisface principiul optimalității este și tehnica greedy. În Secțiunea 8.6 vom ilustra comparativ aceste două tehnici.

Ca și în cazul algoritmilor greedy, soluția optimă nu este în mod necesar unică. Dezvoltarea unui algoritm de programare dinamică poate fi descrisă de următoarea succesiune de pași:

- se caracterizează structura unei soluții optime
- se definește recursiv valoarea unei soluții optime
- se calculează de jos în sus valoarea unei soluții optime

Dacă pe lângă valoarea unei soluții optime se dorește și soluția propriu-zisă, atunci se mai efectuează următorul pas:

- din informațiile calculate se construiește de sus în jos o soluție optimă

Acest pas se rezolvă în mod natural printr-un algoritm recursiv, care efectuează o parcurgere în sens invers a secvenței optime de decizii calculate anterior prin algoritmul de programare dinamică.

8.2 O competiție

În acest prim exemplu de programare dinamică nu ne vom concentra pe principiul optimalității, ci pe structura de control și pe ordinea rezolvării subcazurilor. Din această cauză, problema considerată în această secțiune nu va fi o problemă de optimizare.

Să ne imaginăm o competiție în care doi jucători A și B joacă o serie de cel mult $2n-1$ partide, câștigător fiind jucătorul care acumulează primul n victorii. Presupunem că nu există partide egale, că rezultatele partidelor sunt independente între ele și că pentru orice partidă există o probabilitate p constantă ca să câștige jucătorul A și o probabilitate $q = 1-p$ ca să câștige jucătorul B.

Ne propunem să calculăm $P(i, j)$, probabilitatea ca jucătorul A să câștige competiția, dat fiind că mai are nevoie de i victorii și că jucătorul B mai are nevoie de j victorii pentru a câștiga. În particular, la începutul competiției această probabilitate este $P(n, n)$, deoarece fiecare jucător are nevoie de n victorii. Pentru $1 \le i \le n$, avem $P(0, i) = 1$ și $P(i, 0) = 0$. Probabilitatea $P(0, 0)$ este nedefinită. Pentru $i, j \ge 1$, putem calcula $P(i, j)$ după formula:

$$P(i, j) = pP(i-1, j) + qP(i, j-1)$$

algoritmul corespunzător fiind:

```
function P(i, j)
    if i = 0 then return 1
    if j = 0 then return 0
    return pP(i−1, j) + qP(i, j−1)
```

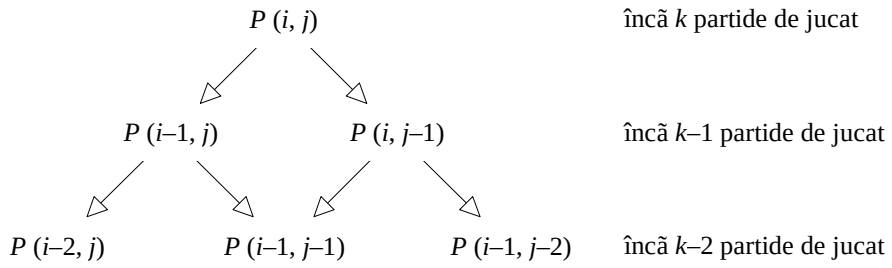


Figura 8.1 Apelurile recursive efectuate după un apel al funcției $P(i, j)$.

Fie $t(k)$ timpul necesar, în cazul cel mai nefavorabil, pentru a calcula probabilitatea $P(i, j)$, unde $k = i+j$.

Avem:

$$t(1) \le a$$

$$t(k) \le 2t(k-1) + c, \quad k > 1$$

a și c fiind două constante. Prin metoda iterației, obținem $t \in O(2^k)$, iar dacă $i = j = n$, atunci $t \in O(4^n)$. Dacă urmărim modul în care sunt generate apelurile recursive (Figura 8.1), observăm că este identic cu cel pentru calculul ineficient al coeficienților binomiali:

$$C(i+j, j) = C((i-1)+j, j) + C(i+(j-1), j-1)$$

Din Exercițiul 8.1 rezultă că numărul total de apeluri recursive este

$$2\binom{i+j}{j} - 2$$

Timpul de execuție pentru un apel $P(n, n)$ este deci în $\Omega(\binom{2n}{n})$. Ținând cont și de Exercițiul 8.3, obținem că timpul pentru calculul lui $P(n, n)$ este în $O(4^n) \cap \Omega(4^n/n)$. Aceasta înseamnă că, pentru valori mari ale lui n, algoritmul este ineficient.

Pentru a îmbunătăți algoritmul, vom proceda ca în cazul triunghiului lui Pascal. Tabloul în care memorăm rezultatele intermediare nu îl vom completa, însă, linie cu linie, ci pe diagonală. Probabilitatea $P(n, n)$ poate fi calculată printr-un apel *serie*(n, p) al algoritmului

```
function serie(n, p)
    array P[0..n, 0..n]
    q ← 1−p
    for s ← 1 to n do
        P[0, s] ← 1; P[s, 0] ← 0
        for k ← 1 to s−1 do
            P[k, s−k] ← pP[k−1, s−k] + qP[k, s−k−1]
    for s ← 1 to n do
        for k ← 0 to n−s do
            P[s+k, n−k] ← pP[s+k−1, n−k] + qP[s+k, n−k−1]
    return P[n, n]
```

Deoarece în esență se completează un tablou de $n \times n$ elemente, timpul de execuție pentru un apel *serie*(n, p) este în $\Theta(n^2)$. Ca și în cazul coeficienților binomiali, nu este nevoie să memorăm întregul tablou P. Este suficient să memorăm diagonala curentă din P, într-un vector de n elemente.

8.3 Înmulțirea înlănțuită a matricilor

Ne propunem să calculăm produsul matricial

$$M = M_1 M_2 \ldots M_n$$

Deoarece înmulțirea matricilor este asociativă, putem opera aceste înmulțiri în mai multe moduri. Înainte de a considera un exemplu, să observăm că înmulțirea clasică a unei matrici de $p \times q$ elemente cu o matrice de $q \times r$ elemente necesită pqr înmulțiri scalare.

Dacă dorim să obținem produsul $ABCD$ al matricilor A de 13×5, B de 5×89, C de 89×3 și D de 3×34 elemente, în funcție de ordinea efectuării înmulțirilor matriciale (dată prin paranteze), numărul total de înmulțiri scalare poate să fie foarte diferit:

$(((AB)C)D)$	10582	înmulțiri
$((AB)(CD))$	54201	înmulțiri
$((A(BC))D)$	2856	înmulțiri
$(A((BC)D))$	4055	înmulțiri
$(A(B(CD)))$	26418	înmulțiri

Cea mai eficientă metodă este de aproape 19 ori mai rapidă decât cea mai ineficientă. În concluzie, ordinea de efectuare a înmulțirilor matriciale poate avea un impact dramatic asupra eficienței.

În general, vom spune că un produs de matrici este *complet parantezat*, dacă este: *i)* o singură matrice, sau *ii)* produsul a două produse de matrici complet parantezate, înconjurat de paranteze. Pentru a afla în mod direct care este ordinea optimă de efectuare a înmulțirilor matriciale, ar trebui să parantezăm expresia lui M în toate modurile posibile și să calculăm de fiecare dată care este numărul de înmulțiri scalare necesare.

Să notăm cu $T(n)$ numărul de moduri în care se poate paranteza complet un produs de n matrici. Să presupunem că decidem să facem prima "tăietură" între a i-a și a $(i+1)$-a matrice a produsului

$$M = (M_1\, M_2\, \ldots\, M_i)(M_{i+1}\, M_{i+2}\, \ldots\, M_n)$$

Sunt acum $T(i)$ moduri de a paranteza termenul stâng și $T(n-i)$ moduri de a paranteza termenul drept. Deoarece i poate lua orice valoare între 1 și $n-1$, obținem recurența

$$T(n) = \sum_{i=1}^{n-1} T(i)\; T(n-i)$$

cu $T(1) = 1$. De aici, putem calcula toate valorile lui $T(n)$. De exemplu, $T(5) = 14$, $T(10) = 4862$, $T(15) = 2674440$. Valorile lui $T(n)$ sunt cunoscute ca *numerele catalane*. Se poate demonstra că

$$T(n) = \frac{1}{n}\binom{2n-2}{n-1}$$

Din Exercițiul 8.3 rezultă $T \in \Omega(4^n/n^2)$. Deoarece, pentru fiecare mod de parantezare, operația de numărare a înmulțirilor scalare necesită un timp în $\Omega(n)$, determinarea modului optim de a-l calcula pe M este în $\Omega(4^n/n)$. Această metodă directă este deci foarte neperformantă și o vom îmbunătăți în cele ce urmează.

Din fericire, principiul optimalității se poate aplica la această problemă. De exemplu, dacă cel mai bun mod de a înmulți toate matricile presupune prima tăietură între a i-a și a $i+1$-a matrice a produsului, atunci subprodusele $M_1\, M_2\, \ldots\, M_i$ și $M_{i+1}\, M_{i+2}\, \ldots\, M_n$ trebuie și ele calculate într-un mod optim. Aceasta ne sugerează să aplicăm programarea dinamică.

Vom construi tabloul $m[1\,..\,n,\, 1\,..\,n]$, unde $m[i, j]$ este numărul minim de înmulțiri scalare necesare pentru a calcula partea $M_i\, M_{i+1}\, \ldots\, M_j$ a produsului inițial. Soluția problemei inițiale va fi dată de $m[1, n]$. Presupunem că tabloul $d[0\,..\,n]$ conține dimensiunile matricilor M_i, astfel încât matricea M_i este de dimensiune $d[i-1] \times d[i]$, $1 \le i \le n$. Construim tabloul m diagonală cu diagonală: diagonala s conține elementele $m[i, j]$ pentru care $j - i = s$. Obținem astfel succesiunea

$s = 0$: $m[i, i] = 0, \quad i=1, 2, \ldots, n$
$s = 1$: $m[i, i+1] = d[i-1]\ d[i]\ d[i+1], \quad i=1, 2, \ldots, n-1$
$1 < s < n$: $m[i, i+s] = \min_{i \le k < i+s} (m[i, k] + m[k+1, i+s] + d[i-1]\ d[k]\ d[i+s])$,
$i = 1, 2, \ldots, n-s$

A treia situație reprezintă faptul că, pentru a calcula $M_i\ M_{i+1}\ \ldots\ M_{i+s}$, încercăm toate posibilitățile

$$(M_i\ M_{i+1}\ \ldots\ M_k)\ (M_{k+1}\ M_{k+2}\ \ldots\ M_{i+s})$$

și o alegem pe cea optimă, pentru $i \le k < i+s$. A doua situație este de fapt o particularizare a celei de-a treia situații, cu $s = 1$.

Pentru matricile A, B, C, D, din exemplul precedent, avem

$$d = (13, 5, 89, 3, 34)$$

Pentru $s = 1$, găsim $m[1, 2] = 5785$, $m[2, 3] = 1335$, $m[3, 4] = 9078$. Pentru $s = 2$, obținem

$m[1, 3] = \min(m[1, 1] + m[2, 3] + 13\times5\times3,\ m[1, 2] + m[3, 3] + 13\times89\times3)$
$= \min(1530, 9256) = 1530$

$m[2, 4] = \min(m[2, 2] + m[3, 4] + 5\times89\times34,\ m[2, 3] + m[4, 4] + 5\times3\times34)$
$= \min(24208, 1845) = 1845$

Pentru $s = 3$,

$m[1, 4] = \min(\ \{k = 1\}\ m[1, 1] + m[2, 4] + 13\times5\times34,$
$\{k = 2\}\ m[1, 2] + m[3, 4] + 13\times89\times34,$
$\{k = 3\}\ m[1, 3] + m[4, 4] + 13\times3\times34)$
$= \min(4055, 54201, 2856) = 2856$

Tabloul m este dat în Figura 8.2.

Să calculăm acum eficiența acestei metode. Pentru $s > 0$, sunt $n-s$ elemente de calculat pe diagonala s; pentru fiecare, trebuie să alegem între s posibilități (diferite valori posibile ale lui k). Timpul de execuție este atunci în ordinul exact al lui

$$\sum_{s=1}^{n-1}(n-s)s = n\sum_{s=1}^{n-1}s - \sum_{s=1}^{n-1}s^2 = n^2(n-1)/2 - n(n-1)(2n-1)/6 = (n^3-n)/6$$

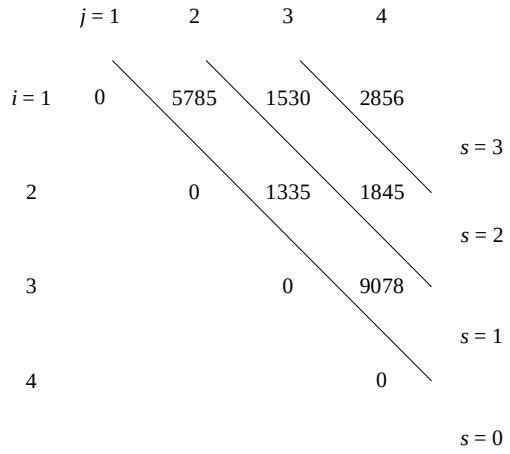


Figura 8.2 Exemplu de înmulțire înlănțuită a unor matrici.

Timpul de execuție este deci în $\Theta(n^3)$, ceea ce reprezintă un progres remarcabil față de metoda exponențială care verifică toate parantezările posibile*.

Prin această metodă, îl putem afla pe $m[1, n]$. Pentru a determina și cum să calculăm produsul M în cel mai eficient mod, vom mai construi un tablou $r[1 .. n, 1 .. n]$, astfel încât $r[i, j]$ să conțină valoarea lui k pentru care este obținută valoarea minimă a lui $m[i, j]$. Următorul algoritm construiește tablourile globale m și r.

```
procedure minscal(d[0 .. n])
    for i ← 1 to n do m[i, i] ← 0
    for s ← 1 to n−1 do
        for i ← 1 to n−s do
            m[i, i+s] ← +∞
            for k ← i to i+s−1 do
                q ← m[i, k] + m[k+1, i+s] + d[i−1] d[k] d[i+s]
                if q < m[i, i+s] then   m[i, i+s] ← q
                                        r[i, i+s] ← k
```

Produsul M poate fi obținut printr-un apel *minmat*(1, n) al algoritmului recursiv

---

* Problema înmulțirii înlănțuite optime a matricilor poate fi rezolvată și prin algoritmi mai eficienți. Astfel, T. C. Hu și M. R. Shing au propus, (în 1982 și 1984), un algoritm cu timpul de execuție în $O(n \log n)$.

function *minmat*(*i*, *j*)
  {returnează produsul matricial $M_i\, M_{i+1} \ldots M_j$
   calculat prin *m*[*i*, *j*] înmulțiri scalare;
   se presupune că $i \le r[i, j] \le j$}
  if $i = j$ **then return** M_i
  arrays *U*, *V*
  $U \leftarrow$ *minmat*(*i*, *r*[*i*, *j*])
  $V \leftarrow$ *minmat*(*r*[*i*, *j*]+1, *j*)
  return *produs*(*U*, *V*)

unde funcția *produs*(*U*, *V*) calculează în mod clasic produsul matricilor *U* și *V*. În exemplul nostru, produsul *ABCD* se va calcula în mod optim cu 2856 înmulțiri scalare, corespunzător parantezării: ((*A*(*BC*))*D*).

8.4 Tablouri multidimensionale

Implementarea operațiilor cu matrici și, în particular, a algoritmilor de înmulțire prezentați în Secțiunile 7.8 și 8.3 necesită, în primul rând, clarificarea unor aspecte legate de utilizarea tablourilor în limbajele C și C++.

În privința tablourilor, limbajul C++ nu aduce nimic nou față de următoarele două reguli preluate din limbajul C:

- *Din punct de vedere sintactic, noțiunea de tablou multidimensional nu există.* Regula este surprinzătoare deoarece, în mod cert, putem utiliza tablouri multidimensionale. De exemplu, `int a[2][5]` este un tablou multidimensional (bidimensional) corect definit, având două linii și cinci coloane, iar `a[1][2]` este unul din elementele sale, și anume al treilea de pe a doua linie. Această contradicție aparentă este generată de o ambiguitate de limbaj: prin `int a[2][5]` am definit, de fapt, două tablouri de câte cinci elemente. Altfel spus, `a` este un tablou de tablouri și, ca o primă consecință, rezultă că numărul dimensiunilor unui "tablou multidimensional" este nelimitat. O altă consecință este chiar modalitatea de memorare a elementelor. Așa cum este normal, cele două tablouri (de câte cinci elemente) din `a` sunt memorate într-o zonă continuă de memorie, unul după altul. Deci, elementele tablourilor bidimensionale sunt memorate pe linii. În general, elementele tablourilor multidimensionale sunt memorate astfel încât ultimul indice variază cel mai rapid.
- *Un identificator de tablou este, în același timp, un pointer a cărui valoare este adresa primului element al tabloului.* Prin această regulă, tablourile sunt identificate cu adresele primelor lor elemente. De exemplu, identificatorul `a` de

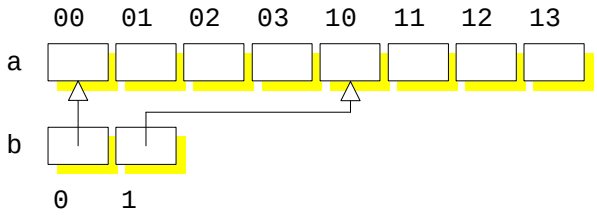


Figura 8.3 Structura zonelor de memorie de la adresele `a` și `b`.

mai sus (definit ca `int a[2][5]`) este de tip pointer la un tablou cu cinci elemente întregi, adică `int (*)[5]`, iar `a[0]` și `a[1]` sunt adrese de întregi, adică `int*`. Mai exact, expresia `a[0]` este adresa primei linii din matrice (a primului tablou de cinci elemente) și este echivalentă cu `*(a+0)`, iar expresia `a[1]` este adresa celei de-a doua linii din matrice (a celui de-al doilea tablou de cinci elemente), adică `*(a+1)`. În final, deducem că `a[1][2]` este echivalent cu `*(*(a+1)+2)`, ceea ce ilustrează echivalența operatorului de indexare și a celui de indirectare.

În privința echivalenței identificatorilor de tablouri și a pointerilor, nu mai putem fi atât de categorici. Să pornim de la următoarele două definiții:

```
int a[ 2 ][ 5 ];
int *b[ 2 ] = {
                a[ 0 ]    // adica b[ 0 ] = &a[ 0 ][ 0 ]
                a[ 1 ]    // adica b[ 1 ] = &a[ 1 ][ 0 ]
              };
```

unde `a` este un tablou de 2×5 elemente întregi, iar `b` este un tablou de două adrese de întregi. Structura zonelor de memorie de la adresele `a` și `b` este prezentată în Figura 8.3.

Evaluând expresia `b[1][2]`, obținem `*(*(b+1)+2)`, adică elementul `a[1][2]`, element adresat și prin expresia echivalentă `*(*(a+1)+2)`. Se observă că valoarea pointerului `*(b+1)` este memorată în al doilea element din `b` (de adresă `b+1`), în timp ce valoarea `*(a+1)`, tot de tip pointer la `int`, nu este memorată, fiind substituită direct cu adresa celei de-a doua linii din `a`. Pentru sceptici, programul următor ilustrează aceste afirmații.

```
#include <iostream.h>

main( ) {
  int a[ 2 ][ 5 ];
  int *b[ 2 ] = { a[ 0 ], a[ 1 ] };

  cout << ( a + 1 ) << ' ' << *( a + 1 ) << '\n';
  cout << ( b + 1 ) << ' ' << *( b + 1 ) << '\n';

  return 1;
}
```

Tratarea diferită a expresiilor echivalente `*(b+1)` și `*(a+1)` se datorează faptului că identificatorii de tablouri nu sunt de tip pointer, ci de tip pointer constant. Valoarea lor nu poate fi modificată, deoarece este o constantă rezultată în urma compilării programului. Astfel, dacă definim

```
char x[ ] = "algoritm";
char *y   = "eficient";
```

atunci `x` este adresa unei zone de memorie care conține textul “algoritm”, iar `y` este adresa unei zone de memorie care conține adresa șirului “eficient”.

Expresiile `x[1]`, `*(x+1)` și expresiile `y[1]`, `*(y+1)` sunt corecte, valoarea lor fiind al doilea caracter din șirurile “algoritm” și, respectiv, “eficient”. În schimb, dintre cele două expresii `*(++x)` și `*(++y)`, doar a doua este corectă, deoarece valoarea lui `x` nu poate fi modificată.

Prin introducerea claselor și prin posibilitatea de supraîncărcare a operatorului `[]`, echivalența dintre operatorul de indirectare `*` și cel de indexare `[]` nu mai este valabilă. Pe baza definiției

```
int D = 8192;
// ...
tablou<int> x( D );
```

putem scrie oricând

```
for ( int i = 0; i < D; i++ ) x[ i ] = i;
```

dar nu și

```
for ( i = 0; i < D; i++ ) *( x + i ) = i;
```

deoarece expresia `x+i` nu poate fi calculată. Cu alte cuvinte, identificatorii de tip `tablou<T>` nu mai sunt asimilați tipului pointer. Într-adevăr, identificatorul `x`, definit ca `tablou<float> x( D )`, nu este identificatorul unui tablou predefinit,

ci al unui tip definit utilizator, tip care, întâmplător, are un comportament de tablou. Dacă totuși dorim ca expresia `*(x+i)` să fie echivalentă cu `x[i]`, nu avem decât să definim în clasa `tablou<T>` operatorul

```
template <class T>
T* operator +( tablou<T>& t, int i ) {
  return &t[ i ];
}
```

În continuare, ne întrebăm dacă avem posibilitatea de a defini tablouri multidimensionale prin clasa `tablou<T>`, fără a introduce un tip nou. Răspunsul este afirmativ și avem două variante de implementare:

- Orice clasă permite definirea unor tablouri de obiecte. În particular, pentru clasa `tablou<T>`, putem scrie

  ```
  tablou<int> c[ 3 ];
  ```

  ceea ce înseamnă că `c` este un tablou de trei elemente de tip `tablou<int>`. Inițializarea acestor elemente se realizează prin specificarea explicită a argumentelor constructorilor.

  ```
  tablou<int> x( 5 );  // un tablou de 5 de elmente
  tablou<int> c[ 3 ] = { tablou<int>( x ),
                         tablou<int>( 9 )
                       };
  ```

  În acest exemplu, primul element se inițializează prin constructorul de copiere, al doilea prin constructorul cu un singur argument `int` (numărul elementelor), iar al treilea prin constructorul implicit. În expresia `c[1][4]`, care se referă la al cincilea element din cea de-a doua linie, primul operator de indexare folosit este cel predefinit, iar al doilea este cel supraîncărcat în clasa `tablou<T>`. Din păcate, `c` este în cele din urmă tot un tablou predefinit, având deci toate deficiențele menționate în Secțiunea 4.1. În particular, este imposibil de verificat corectitudinea primului indice, în timp ce verificarea celui de-al doilea poate fi activată selectiv, pentru fiecare linie.
- O a doua modalitate de implementare a tablourilor multidimensionale utilizează din plin facilitățile claselor parametrice. Prin instrucțiunea

  ```
  tablou< tablou<int> > d( 3 );
  ```

  obiectul `d` este definit ca un tablou cu trei elemente, fiecare element fiind un tablou de `int`.

  Problema care apare aici este cum să dimensionăm cele trei tablouri membre, tablouri inițializate prin constructorul implicit. Nu avem nici o modalitate de a specifica argumentele constructorilor (ca și în cazul alocării tablourilor prin

operatorul `new`), unica posibilitate rămânând atribuirea explicită sau funcția de modificare a dimensiunii (redimensionare).

```
tablou<int> x( 25 );
tablou< tablou<int> > d( 3 );
d[ 0 ] = x;           // prima linie se initializeaza cu x
d[ 1 ].newsize( 16 ); // a doua linie se redimensioneaza
                      // a treia linie nu se modifica
```

Adresarea elementelor tabloului `d` constă în evaluarea expresiilor de genul `d[1][4]`, unde operatorii de indexare `[]` sunt, de această dată, ambii din clasa parametrică `tablou<T>`. În consecință, activarea verificărilor de indici poate fi invocată fie prin `d.vOn()`, pentru indicele de linie, fie separat în fiecare linie, prin `d[i].vOn()`, pentru cel de coloană.

În anumite situații, tablourile multidimensionale definite prin clasa parametrică `tablou<T>` au un avantaj important față de cele predefinite, în ceea ce privește consumul de memorie. Pentru fixarea ideilor, să considerăm tablouri bidimensionale, adică *matrici*. Dacă liniile unei matrici nu au același număr de elemente, atunci:

- În tablourile predefinite, fiecare linie este de lungime maximă.
- În tablourile bazate pe clasa `tablou<T>`, fiecare linie poate fi dimensionată corespunzător numărului efectiv de elemente.

O matrice este *triunghiulară*, atunci când doar elementele situate de-o parte a diagonalei principale* sunt efectiv utilizate. În particular, o matrice triunghiulară este *inferior triunghiulară*, dacă folosește numai elementele de sub diagonala principală și *superior trunghiulară*, în caz contrar. Matricile trunghiulare au deci nevoie numai de aproximativ jumătate din spațiul necesar unei matrici obișnuite.

Tablourile bazate pe clasa `tablou<T>` permit implementarea matricilor triunghiulare în spațiul strict necesar, prin dimensionarea corespunzătoare a fiecărei linii. Pentru tablourile predefinite, acest lucru este posibil doar prin utilizarea unor artificii de calcul la adresarea elementelor.

8.5 Determinarea celor mai scurte drumuri într-un graf

Fie $G = <V, M>$ un graf orientat, unde V este mulțimea vârfurilor și M este mulțimea muchiilor. Fiecărei muchii i se asociază o lungime nenegativă. Dorim să calculăm lungimea celui mai scurt drum între fiecare pereche de vârfuri.

* *Diagonala principală* este diagonala care unește colțul din stânga sus cu cel din dreapta jos.

Vom presupune că vârfurile sunt numerotate de la 1 la n și că matricea L dă lungimea fiecărei muchii: $L[i, i] = 0$, $L[i, j] \geq 0$ pentru $i \neq j$, $L[i, j] = +\infty$ dacă muchia (i, j) nu există.

Principiul optimalității este valabil: dacă cel mai scurt drum de la i la j trece prin vârful k, atunci porțiunea de drum de la i la k, cât și cea de la k la j, trebuie să fie, de asemenea, optime.

Construim o matrice D care să conțină lungimea celui mai scurt drum între fiecare pereche de vârfuri. Algoritmul de programare dinamică inițializează pe D cu L. Apoi, efectuează n iterații. După iterația k, D va conține lungimile celor mai scurte drumuri care folosesc ca vârfuri intermediare doar vârfurile din $\{1, 2, \ldots, k\}$. După n iterații, obținem rezultatul final. La iterația k, algoritmul trebuie să verifice, pentru fiecare pereche de vârfuri (i, j), dacă există sau nu un drum, trecând prin vârful k, care este mai bun decât actualul drum optim ce trece doar prin vârfurile din $\{1, 2, \ldots, k-1\}$. Fie D_k matricea D după iterația k. Verificarea necesară este atunci:

$$D_k[i, j] = \min(D_{k-1}[i, j], D_{k-1}[i, k] + D_{k-1}[k, j])$$

unde am făcut uz de principiul optimalității pentru a calcula lungimea celui mai scurt drum via k. Implicit, am considerat că un drum optim care trece prin k nu poate trece de două ori prin k.

Acest algoritm simplu este datorat lui Floyd (1962):

```
function Floyd(L[1 .. n, 1 .. n])
    array D[1 .. n, 1 .. n]
    D ← L
    for k ← 1 to n do
        for i ← 1 to n do
            for j ← 1 to n do
                D[i, j] ← min(D[i, j], D[i, k]+D[k, j])
    return D
```

De exemplu, dacă avem

$$D_0 = L = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & \infty & 0 & 15 \\ 15 & \infty & 5 & 0 \end{pmatrix}$$

obținem succesiv

$$D_1 = \begin{pmatrix} 0 & 5 & \infty & \infty \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix} \qquad D_2 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 50 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

$$D_3 = \begin{pmatrix} 0 & 5 & 20 & 10 \\ 45 & 0 & 15 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix} \qquad D_4 = \begin{pmatrix} 0 & 5 & 15 & 10 \\ 20 & 0 & 10 & 5 \\ 30 & 35 & 0 & 15 \\ 15 & 20 & 5 & 0 \end{pmatrix}$$

Puteți deduce că algoritmul lui Floyd necesită un timp în $\Theta(n^3)$. Un alt mod de a rezolva această problemă este să aplicăm algoritmul *Dijkstra* (Capitolul 6) de n ori, alegând mereu un alt vârf sursă. Se obține un timp în $n\,\Theta(n^2)$, adică tot în $\Theta(n^3)$. Algoritmul lui Floyd, datorită simplității lui, are însă constanta multiplicativă mai mică, fiind probabil mai rapid în practică. Dacă folosim algoritmul *Dijkstra-modificat* în mod similar, obținem un timp total în $O(\max(mn, n^2) \log n)$, unde $m = \#M$. Dacă graful este rar, atunci este preferabil să aplicăm algoritmul *Dijkstra-modificat* de n ori; dacă graful este dens ($m \cong n^2$), este mai bine să folosim algoritmul lui Floyd.

De obicei, dorim să aflăm nu numai lungimea celui mai scurt drum, dar și traseul său. În acestă situație, vom construi o a doua matrice P, inițializată cu zero. Bucla cea mai interioară a algoritmului devine

if $D[i, k]+D[k, j] < D[i, j]$ **then** $D[i, j] \leftarrow D[i, k]+D[k, j]$
$P[i, j] \leftarrow k$

Când algoritmul se oprește, $P[i, j]$ va conține vârful din ultima iterație care a cauzat o modificare în $D[i, j]$. Pentru a afla prin ce vârfuri trece cel mai scurt drum de la i la j, consultăm elementul $P[i, j]$. Dacă $P[i, j] = 0$, atunci cel mai scurt drum este chiar muchia (i, j). Dacă $P[i, j] = k$, atunci cel mai scurt drum de la i la

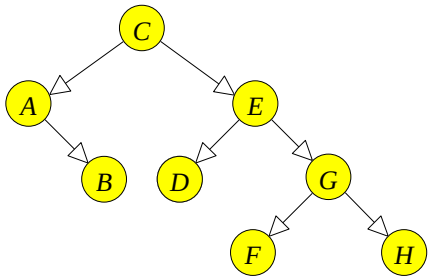


Figura 8.4 Un arbore binar de căutare.

j trece prin *k* și urmează să consultăm recursiv elementele *P*[*i*, *k*] și *P*[*k*, *j*] pentru a găsi și celelalte vârfuri intermediare.

Pentru exemplul precedent se obține

$$P = \begin{pmatrix} 0 & 0 & 4 & 2 \\ 4 & 0 & 4 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

Deoarece *P*[1, 3] = 4, cel mai scurt drum de la 1 la 3 trece prin 4. Deoarece *P*[1, 4] = 2, cel mai scurt drum de la 1 la 4 trece prin 2. Rezultă că cel mai scurt drum de la 1 la 3 este: 1, 2, 4, 3.

8.6 Arbori binari optimi de căutare

Un arbore binar în care fiecare vârf conține o valoare (numită *cheie*) este un *arbore de căutare*, dacă cheia fiecărui vârf neterminal este mai mare sau egală cu cheile descendenților săi stângi și mai mică sau egală cu cheile descendenților săi drepți. Dacă cheile arborelui sunt distincte, aceste inegalități sunt, în mod evident, stricte.

Figura 8.4 este un exemplu de arbore de căutare*, conținând cheile *A*, *B*, *C*, …, *H*. Vârfurile pot conține și alte informații (în afară de chei), la care să avem acces prin intermediul cheilor.

* În această secțiune vom subînțelege că toți arborii de căutare sunt binari.

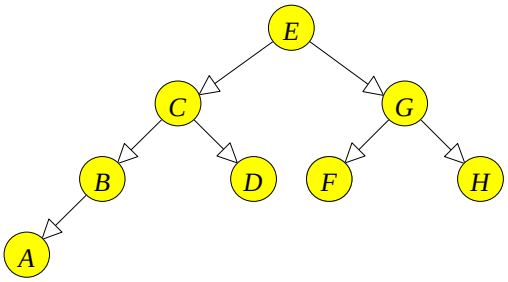


Figura 8.5 Un alt arbore binar de căutare.

Această structură de date este utilă, deoarece permite o căutare eficientă a valorilor în arbore (Exercițiul 8.10). De asemenea, este posibil să actualizăm un arbore de căutare (să ștergem un vârf, să modificăm valoarea unui vârf, sau să adăugăm un vârf) într-un mod eficient, fără să distrugem proprietatea de arbore de căutare.

Cu o mulțime dată de chei, se pot construi mai mulți arbori de căutare (Figura 8.5).

Pentru a căuta o cheie X în arborele de căutare, X va fi comparată la început cu cheia rădăcinii arborelui. Dacă X este mai mică decât cheia rădăcinii, atunci se continuă căutarea în subarborele stâng; dacă X este egală cu cheia rădăcinii, atunci căutarea se încheie cu succes; dacă X este mai mare decât cheia rădăcinii, atunci se continuă căutarea în subarborele drept. Se continuă apoi recursiv acest proces.

De exemplu, în arborele din Figura 8.4 putem găsi cheia E prin două comparații, în timp ce aceeași cheie poate fi găsită în arborele din Figura 8.5 printr-o singură comparație. Dacă cheile A, B, C, …, H au aceeași probabilitate, atunci pentru a găsi o cheie oarecare sunt necesare în medie:

(2+3+1+3+2+4+3+4)/8 = 22/8 comparații, pentru arborele din Figura 8.4

(4+3+2+3+1+3+2+3)/8 = 21/8 comparații, pentru arborele din Figura 8.5

Când cheile sunt echiprobabile, arborele de căutare care minimizează numărul mediu de comparații necesare este arborele de căutare de înălțime minimă (demonstrați acest lucru și găsiți o metodă pentru a construi arborele respectiv!).

Vom rezolva în continuare o problemă mai generală. Să presupunem că avem cheile $c_1 < c_2 < \ldots < c_n$ și că, în tabloul p, $p[i]$ este probabilitatea cu care este căutată cheia c_i, $1 \leq i \leq n$. Pentru simplificare, vom considera că sunt căutate doar cheile prezente în arbore, deci că $p[1]+p[2]+\ldots+p[n] = 1$. Ne propunem să găsim arborele optim de căutare pentru cheile c_1, c_2, …, c_n, adică arborele care minimizează numărul mediu de comparații necesare pentru a găsi o cheie.

Problema este similară cu cea a găsirii arborelui cu lungimea externă ponderată minimă (Secțiunea 6.3), cu deosebirea că, de această dată, trebuie să menținem ordinea cheilor. Această restricție face ca problema găsirii arborelui optim de căutare să fie foarte asemănătoare cu problema înmulțirii înlănțuite a matricilor. În esență, se poate aplica același algoritm.

Dacă o cheie c_i se află într-un vârf de adîncime d_i, atunci sunt necesare d_i+1 comparații pentru a o găsi. Pentru un arbore dat, numărul mediu de comparații necesare este

$$\sum_{i=1}^{n} p[i](d_i+1)$$

Dorim să găsim arborele pentru care acest număr este minim.

Vom rezolva această problemă prin metoda programării dinamice. Prima decizie constă în a determina cheia c_k a rădăcinii. Să observăm că este satisfăcut principiul optimalității: dacă avem un arbore optim pentru $c_1, c_2, \ldots, c_n$ și cu cheia c_k în rădăcină, atunci subarborii săi stâng și drept sunt arbori optimi pentru cheile $c_1, c_2, \ldots, c_{k-1}$, respectiv $c_{k+1}, c_{k+2}, \ldots, c_n$. Mai general, într-un arbore optim conținând cele n chei, un subarbore oarecare este la rândul său optim pentru o secvență de chei succesive $c_i, c_{i+1}, \ldots, c_j$, $i \le j$.

În tabloul C, să notăm cu $C[i, j]$ numărul mediu de comparații efectuate într-un subarbore care este optim pentru cheile $c_i, c_{i+1}, \ldots, c_j$, atunci când se caută o cheie X în arborele optim principal. Valoarea

$$m[i, j] = p[i] + p[i+1] + \ldots + p[j]$$

este probabilitatea ca X să se afle în secvența $c_i, c_{i+1}, \ldots, c_j$. Fie c_k cheia rădăcinii subarborelui considerat. Atunci, probabilitatea comparării lui X cu c_k este $m[i, j]$, și avem:

$$C[i, j] = m[i, j] + C[i, k-1] + C[k+1, j]$$

Pentru a obține schema de programare dinamică, rămîne să observăm că c_k (cheia rădăcinii subarborelui) este aleasă astfel încât

$$C[i, j] = m[i, j] + \min_{i \le k \le j} (C[i, k-1]+C[k+1, j]) \qquad (*)$$

În particular, $C[i, i] = p[i]$ și $C[i, i-1] = 0$.

Dacă dorim să găsim arborele optim pentru cheile $c_1 < c_2 < \ldots < c_5$, cu probabilitățile

$p[1] = 0{,}30$ $p[2] = 0{,}05$ $p[3] = 0{,}08$
$p[4] = 0{,}45$ $p[5] = 0{,}12$

calculăm pentru început matricea *m*:

$$m = \begin{pmatrix} 0{,}30 & 0{,}35 & 0{,}43 & 0{,}88 & 1{,}00 \\ & 0{,}05 & 0{,}13 & 0{,}58 & 0{,}70 \\ & & 0{,}08 & 0{,}53 & 0{,}65 \\ & & & 0{,}45 & 0{,}57 \\ & & & & 0{,}12 \end{pmatrix}$$

Să notăm că $C[i, i] = p[i]$, $1 \le i \le 5$. Din relația (*), calculăm celelalte valori pentru $C[i, j]$:

$C[1, 2] = m[1, 2] + \min(C[1, 0]+C[2, 2], C[1, 1]+C[3, 2])$
$= 0{,}35 + \min(0{,}05, 0{,}30) = 0{,}40$

Similar,

$C[2, 3] = 0{,}18$ $C[3, 4] = 0{,}61$ $C[4, 5] = 0{,}69$

Apoi,

$C[1, 3] = m[1, 3] + \min(C[1, 0]+C[2, 3], C[1, 1]+C[3, 3], C[1, 2]+C[4, 3])$
$= 0{,}43 + \min(0{,}18, 0{,}38, 0{,}40) = 0{,}61$

$C[2, 4] = 0{,}76$ $C[3, 5] = 0{,}85$

$C[1, 4] = 1{,}49$ $C[2, 5] = 1{,}00$

$C[1, 5] = m[1, 5] + \min(C[1, 0]+C[2, 5], C[1, 1]+C[3, 5], C[1, 2]+C[4, 5],$
$C[1, 3]+C[5, 5], C[1, 4]+C[6, 5]) = 1{,}73$

Arborele optim necesită deci în medie 1,73 comparații pentru a găsi o cheie.

În acest algoritm, calculăm valorile $C[i, j]$ în primul rând pentru $j–i = 1$, apoi pentru $j–i = 2$ etc. Când $j–i = q$, avem de calculat $n–q$ valori ale lui $C[i, j]$, fiecare implicând o alegere între $q+1$ posibilități. Timpul necesar* este deci în

$$\Theta(\sum_{q=1}^{n-1} (n-q)(q+1)) = \Theta(n^3)$$

Știm acum cum să calculăm numărul minim de comparații necesare pentru a găsi o cheie în arborele optim. Mai rămâne să construim efectiv arborele optim. În

---

* Dacă ținem cont de îmbunătățirile propuse de D. E. Knuth (*"Tratat de programarea calculatoarelor. Sortare și căutare"*, Secțiunea 6.2.2), acest algoritm de construire a arborilor optimi de căutare poate fi făcut pătratic.

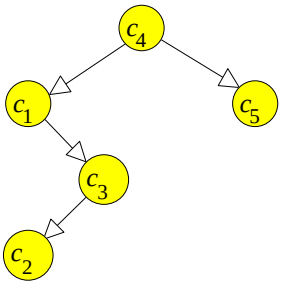


Figura 8.6 Un arbore optim de căutare.

paralel cu tabloul C, vom construi tabloul r, astfel încât $r[i, j]$ să conțină valoarea lui k pentru care este obținută în relația (*) valoarea minimă a lui $C[i, j]$, unde $i < j$. Generăm un arbore binar, conform următoarei metode recursive:

- rădăcina este etichetată cu $(1, n)$
- dacă un vârf este etichetat cu (i, j), $i < j$, atunci fiul său stâng va fi etichetat cu $(i, r[i, j]-1)$ și fiul său drept cu $(r[i, j]+1, j)$
- vârfurile terminale sunt etichetate cu (i, i)

Plecând de la acest arbore, arborele de căutare optim se obține schimbând etichetele (i, j), $i < j$, în $c_{r[i, j]}$, iar etichetele (i, i) în c_i.

Pentru exemplul precedent, obținem astfel arborele optim din Figura 8.6.

Problema se poate generaliza, acceptând să căutăm și chei care nu se află în arbore. Arborele optim de căutare se obține în mod similar.

8.7 Arborii binari de căutare ca tip de dată

Într-o primă aproximare, arborele binar este un tip de dată similar tipului listă. Vârfurile sunt compuse din informație (cheie) și legături, iar arborele propiu-zis este complet precizat prin adresa vârfului rădăcină. În privința organizării memoriei, putem opta fie pentru tablouri paralele, ca în Exercițiul 8.10, fie pentru alocarea dinamică a elementelor. Alegând alocarea dinamică, vom utiliza în întregime modelul oferit de clasa `lista<E>` elaborată în Secțiunea 4.3. Astfel, clasa parametrică `arbore<E>`, cu o structură internă de forma:

```
template <class E>
class arbore {
  // ... declaratii friend
public:
  arbore( )  { root = 0; n = 0; }

  // ... functii membre

private:
  varf<E> *root;  // adresa varfului radacina
  int      n;     // numarul varfurilor din arbore
};
```

are la bază o clasă privată `varf<E>` prin intermediul căreia vom implementa majoritatea operațiilor efectuate asupra arborilor. Vom căuta să izolăm, ori de câte ori va fi posibil, operațiile direct aplicabile vârfurilor, astfel încât interfața dintre cele două clase să fie foarte clar precizată printr-o serie de "operații elementare".

Nu vom implementa în această secțiune arbori binari în toată generalitatea lor, ci doar arborii de căutare. Obiectivul urmărit în prezentarea listelor a fost structura de date în sine, împreună cu procedurile generale de manipulare. În cazul arborelui de căutare, nu mai este necesară o astfel de generalitate, deoarece vom implementa direct operațiile specifice. În mare, aceste operații pot fi împărțite în trei categorii:

- *Căutări*. Localizarea vârfului cu o anumită cheie, a succesorului sau predecesorului lui, precum și a vârfurilor cu cheile de valoare maximă, respectiv minimă.
- *Modificări*. Arborele se modifică prin inserarea sau ștergerea unor vârfuri.
- *Organizări*. Arborele nu este construit prin inserarea elementelor, ci global, stabilind într-o singură trecere legăturile dintre vârfuri. Frecvent, organizarea se face conform unor criterii pentru optimizarea căutărilor. Un caz particular al acestei operații este reorganizarea arborelui după o perioadă suficient de mare de utilizare. Este vorba de reconstruirea arborelui într-o structură optimă, pe baza statisticilor de utilizare.

Datorită operațiilor de căutare și modificare, elementele de tip `E` trebuie să fie comparabile prin operatorii uzuali `==`, `!=`, `>`. În finalul Secțiunii 7.4.1, am arătat că o asemenea pretenție nu este totdeauna justificată. Desigur că, în cazul unor structuri bazate pe relația de ordine, așa cum sunt heap-ul și arborele de căutare, este absolut normal ca elementele să poată fi comparate.

Principalul punct de interes pentru noi este optimizarea, conform algoritmului de programare dinamică. Nu vom ignora nici căutările, nici operațiile de modificare (tratate în Secțiunea 8.7.2).

8.7.1 Arborele optim

Vom rezolva problema obținerii arborelui optim în cel mai simplu caz posibil (din punct de vedere al utilizării, dar nu și în privința programării): arborele deja există și trebuie reorganizat într-un arbore de căutare optim. Având în vedere specificul diferit al operațiilor de organizare față de celelalte operații efectuate asupra grafurilor, am considerat util să încapsulăm optimizarea într-o clasă pe care o vom numi "structură pentru optimizarea arborilor" sau, pe scurt, `s8a`.

Clasa `s8a` este o clasă parametrică privată, asociată clasei `arbore<E>`. Funcționalitatea ei constă în:

i) inițializarea unui tablou cu adresele vârfurilor în ordinea crescătoare a probabilităților cheilor

ii) stabilirea de noi legături între vârfuri astfel încât arborele să fie optim.

Principalul motiv pentru care a fost aleasă această implementare este că sunt necesare doar operații modificare a legăturilor. Deplasarea unui vârf (de exemplu, pentru sortare) înseamnă nu numai deplasarea cheii, ci și a informației asociate. Cum fiecare din aceste elemente pot fi oricât de mari, clasa `s8a` realizează o economie semnificativă de timp și (mai ales) de memorie.

Pentru optimizarea propriu-zisă, am implementat atât algoritmul de programare dinamică, cât și pe cel greedy prezentat în Exercițiul 8.12. Deși algoritmul greedy nu garantează obținerea arborelui optim, el are totuși avantajul că este mai eficient decât algoritmul de programare dinamică din punct de vedere al timpului de execuție și al memoriei utilizate. Invocarea optimizării se realizează din clasa `arbore<E>`, prin secvențe de genul

```
arbore<float> af;

// arborele af se creeaza prin inserarea cheilor
// arborele af se utilizeaza

// pe baza probabilitatilor predefinite si actualizate
//  prin utilizarea arborelui se invoca optimizarea

af.re_prodin( ); // sau af.re_greedy( );
```

unde funcțiile membre `re_greedy()` și `re_prodin()` sunt definte astfel:

```
template <class E>
arbore<E>& arbore<E>::re_greedy( ) {
// reorganizare prin metoda greedy
  s8a<E> opt( root, n );
  root = opt.greedy( );
  return *this;
}
```

```
template <class E>
arbore<E>& arbore<E>::re_prodin( ) {
// reorganziare prin programare dinamica
  s8a<E> opt( root, n );
  root = opt.prodin( );
  return *this;
}
```

După adăugarea tuturor funcțiilor și datelor membre necesare implementării funcțiilor `greedy()` și `prodin()`, clasa `s8a` are următoarea structură:

```
template <class E>
class s8a {   // clasa pentru construirea arborelui optim
  friend class arbore<E>;
private:
  s8a( varf<E> *root, int nn ): pvarf( n = nn ) {
    int i = 0;          // indice in pvarf
    setvarf( i, root ); // setarea elementelor din pvarf
  }

  // initializarea tabloului pvarf cu un arbore deja format
  void setvarf( int&, varf<E>* );

  varf<E>* greedy( ) {  // "optim" prin algoritmul greedy
    return _greedy( 0, n );
  }

  varf<E>* prodin( ) {  // optim prin programare dinamica
    _progDinInit( ); return _progDin( 0, n - 1 );
  }

  // functiile prin care se formeaza efectiv arborele
  varf<E>* _greedy     ( int, int );
  varf<E>* _progDin    ( int, int );
  void     _progDinInit( );   // initializeaza tabloul r

  // date membre
  tablou<varf<E>*> pvarf;  // tabloul adreselor varfurilor
  int n;                   // numarul varfurilor din arbore

  // tabloul indicilor necesar alg. de programare dinamica
  tablou< tablou<int> > r;
};
```

În stabilirea valorilor tablourilor `pvarf` și `r` se pot distinge foarte clar cele două etape ale execuției constructorului clasei `s8a`, etape menționate în Secțiunea 4.2.1. Este vorba de etapa de inițializare (implementată prin lista de inițializare a membrilor) și de etapa de atribuire (implementată prin corpul constructorului). Lista de inițializare asociată constructorului clasei `s8a` conține parametrul necesar dimensionării tabloului `pvarf` pentru cele `n` elemente ale arborelui. Cum este însă inițializat tabloul `r` care nu apare în lista de inițializare? În astfel de cazuri, se invocă automat constructorul implicit (apelabil fără nici un argument) al clasei respective. Pentru clasa `tablou<T>`, constructorul implicit doar inițializează cu `0` datele membre.

Etapa de atribuire a constructorului clasei `s8a`, implementată prin invocarea funcției `setvarf()`, constă în parcurgerea arborelui și memorarea adreselor vârfurilor vizitate în tabloul `pvarf`. Funcția `setvarf()` parcurge pentru fiecare vârf subarborele stâng, apoi memorează adresa vârfului curent și, în final, parcurge subarborele drept. După cum vom vedea în Exercițiul 9.1, acest mod de parcurgere are proprietatea că elementele arborelui sunt parcurse în ordine crescătoare. De fapt, este vorba de o metodă de sortare similară *quicksort*-ului, vârful rădăcină având același rol ca și elementul pivot din *quicksort*.

```
template <class E>
void s8a<E>::setvarf( int& poz, varf<E>* x ) {

  if ( x ) {
    setvarf( poz, x->st );
    pvarf[ poz++ ] = x;
    setvarf( poz, x->dr );

    // anulam toate legaturile elementului x
    x->st = x->dr = x->tata = 0;
  }
}
```

În această funcție, `x->st`, `x->dr` și `x->tata` sunt legăturile vârfului curent `x` către fiul stâng, către cel drept și, respectiv, către vârful tată. În plus față de aceste legături, obiectele de tip `varf<E>` mai conțin cheia (informația) propriu-zisă și un câmp auxiliar pentru probabilitatea vârfului (elementului). În consecință, clasa `varf<E>` are următoarea structură:

```
template <class E>
class varf {
  friend class arbore<E>;
  friend class s8a<E>;

private:
  varf( const E& v, float f = 0 ): key( v )
    { st = dr = tata = 0; p = f; }

  varf<E>   *st; // adresa fiului stang
  varf<E>   *dr; // adresa fiului drept
  varf<E> *tata; // adresa varfului tata

  E         key; // cheia
  float       p; // frecventa utilizarii cheii curente
};
```

Implementarea celor două metode de optimizare a arborelui urmează pas cu pas algoritmul greedy și, respectiv, algoritmul de programare dinamică. Ambele (re)stabilesc legăturile dintre vârfuri printr-un proces recursiv, pornind fie direct de la probabilitățile elementelor, fie de la o matrice (matricea `r`) construită pe baza acestor probabilități. Funcțiile care stabilesc legăturile, adică `_progDin()` și `_greedy()`, sunt următoarele:

```
template <class E>
varf<E>* s8a<E>::_greedy( int m, int M ) {
  // m si M sunt limitele subsecventei curente
  if ( m == M ) return 0;

  // se determina pozitia k a celei mai frecvente chei
  int k;                  float pmax = pvarf[ k = m ]->p;
  for ( int i = m; ++i < M; )
    if ( pvarf[ i ]->p > pmax ) pmax = pvarf[ k = i ]->p;

  // se selecteaza adresa varfului de pe pozitia k
  varf<E> *actual = pvarf[ k ];

  // se construiesc subarborii din stanga si din deapta
  // se initializeaza legatura spre varful tata
  if ( (actual->st = _greedy( m,     k )) != 0 )
    actual->st->tata = actual;
  if ( (actual->dr = _greedy( k + 1, M )) != 0 )
    actual->dr->tata = actual;

  // subarborele curent este gata; se returneaza adresa lui
  return actual;
}
```

```
template <class E>
varf<E>* s8a<E>::_progDin( int i, int j ) {
  // i si j, i <=j, sunt coordonatele radacinii
  //  subarborelui curent in tabloul r
  if ( i > j ) return 0;

  // se selecteaza adresa varfului radacina
  varf<E> *actual = pvarf[ r[ j ][ i ] ];

  if ( i != j ) { // daca nu este un varf frunza ...
    // se construiesc subarborii din stanga si din deapta
    // se initializeaza legatura spre varful tata
    if ( (actual->st = _progDin( i, r[j][i] - 1 )) != 0 )
       actual->st->tata = actual;
    if ( (actual->dr = _progDin( r[j][i] + 1, j )) != 0 )
       actual->dr->tata = actual;
  }

  // subarborele curent este gata; se returneaza adresa lui
  return actual;
}
```

Folosind notațiile introduse în descrierea algoritmului de optimizare prin programare dinamică, funcția `_progDinInit()` construiește matricea `r`, unde `r[i][j]`, `i` < `j`, este indicele în tabloul `pvarf` al adresei vârfului etichetat cu (`i`, `j`). În acest scop, se folosește o altă matrice `C`, unde `C[i][j]`, `i` < `j`, este numărul de comparații efectuate în subarborele optim al cheilor cu indicii `i`, …, `j`. Inițial, `C` este completată cu probabilitățile cumulate ale cheilor de indici `i`, …, `j`.

Se observă că matricile `r` și `C` sunt superior triunghiulare. Totuși, pentru implementare, am preferat să lucrăm cu matrici inferior triunghiulare, adică cu transpusele matricilor `r` și `C`, deoarece adresarea elementelor ar fi fost altfel mai complicată.

```
template <class E>
void s8a<E>::_progDinInit( ) {
  int i, j, d;
 tablou< tablou<float> > C;   // tabloul C este local

  // redimensionarea si initializarea tablourilor C si r
  //  ATENTIE! tablourile C si r sunt TRANSPUSE.
  r.newsize( n );
  C.newsize( n );
  for ( i = 0; i < n; i++ ) {
    r[ i ].newsize( i + 1 ); r[ i ][ i ] = i;
    C[ i ].newsize( i + 1 ); C[ i ][ i ] = pvarf[ i ]->p;
  }
```

```
    // pentru inceput C este identic cu m
    for ( d = 1; d < n; d++ )
      for ( i = 0; (j = i + d) < n; i++ )
        C[ j ][ i ] = C[ j - 1 ][ i ] + C[ j ][ j ];

    // elementele din C se calculeaza pe diagonale
    for ( d = 1; d < n; d++ )
      for ( i = 0; (j = i + d) < n; i++ ) {
        // in calculul minimului dintre C[i][k-1]+C[k+1][j]
        //  consideram mai intai cazurile k=i si k=j in care
        //  avem C[i][i-1] = 0 si C[j+1][j] = 0
        int k; float Cmin;
        if ( C[ j ][ i + 1 ] < C[ j - 1 ][ i ] )
          Cmin = C[ j ][ (k = i) + 1 ];
        else
          Cmin = C[ (k = j) - 1 ][ i ];

        // au mai ramas de testat elementele i+1, ..., j-1
        for ( int l = i + 1; l < j; l++ )
          if ( C[ l - 1 ][ i ] + C[ j ][ l + 1 ] < Cmin )
            Cmin = C[ (k = l) - 1 ][ i ] + C[ j ][ l + 1 ];

        // minimul si pozitia lui sunt stabilite ...
        C[ j ][ i ] += Cmin;
        r[ j ][ i ] = k;
      }
  }
```

8.7.2 Căutarea în arbore

Principala operație efectuată prin intermediul arborilor binari de căutare este regăsirea informației asociate unei anumite chei. Funcția de căutare `search()` are ca argument cheia pe baza căreia se va face căutarea și returnează *false* sau *true*, după cum cheia fost regăsită, sau nu a fost regăsită în arbore. Când căutarea s-a terminat cu succes, valoarea din arbore a cheii regăsite este returnată prin intermediul argumentului de tip referință, pentru a permite consultarea informațiilor asociate.

```
template <class E>
int arbore<E>::search( E& k ) {
  varf<E> *x = _search( root, k );
  if ( !x )   return 0; // element absent
  x->p++;               // actualizarea frecventei
  k = x->key; return 1;
}
```

Actualizarea probabilităților cheilor din arbore, după fiecare operație de căutare, este ceva mai delicată, deoarece impune stabilirea importanței evaluărilor existente în raport cu rezultatele căutărilor. De fapt, este vorba de un proces de

învățare care pornește de la anumite cunoștințe deja acumulate. Problema este de a stabili gradul de importanță al cunoștințelor existente în raport cu cele nou dobândite. Înainte de a prezenta o soluție elementară a acestei probleme, să observăm că algoritmii de optimizare lucrează cu probabilități, dar numai ca ponderi. În consecință, rezultatul optimizării nu se schimbă, dacă în loc de probabilități se folosesc frecvențe absolute.

Fie trei chei ale căror probabilități de căutare au fost estimate inițial la 0,18, 0,65, 0,17. Să presupunem că se dorește optimizarea arborelui de căutare asociat acestor chei, atât pe baza acestor estimări, cât și folosind rezultatele a 1000 de căutări de instruire terminate cu succes*. Dacă fixăm ponderea estimărilor inițiale în raport cu rezultatele instruirii la 5 / 2, atunci vom inițializa membrul `p` (estimarea probabilității cheii curente) din clasa `varf<E>` cu valorile

$$0{,}18 \times 1000 \times (5 / 2) = 450$$
$$0{,}65 \times 1000 \times (5 / 2) = 1625$$
$$0{,}17 \times 1000 \times (5 / 2) = 425$$

Apoi, la fiecare căutare terminată cu success, membrul `p` corespunzător cheii găsite se incrementează cu 1. De exemplu, dacă prima cheie a fost găsită în 247 cazuri, a doua în 412 cazuri și a treia în 341 cazuri, atunci valorile lui `p` folosite la optimizarea arborelui vor fi 697, 2037 și 766. Suma acestor valori este 3500, valoare care corespunde celor 1000 de încercări plus ponderea de $1000 \times (5 / 2) = 2500$ asociată estimării inițiale. Noile probabilități, învățate prin instruire, sunt:

$$697 / 3500 \cong 0{,}20$$
$$2037 / 3500 \cong 0{,}58$$
$$766 / 3500 \cong 0{,}22$$

Pentru verificarea rezultatelor de mai sus, să refacem calculele, lucrând numai cu probabilități. Estimările inițiale ale probabilităților sunt 0,18, 0,65 și 0,17. În urma instruirii, cele trei chei au fost căutate cu probabilitățile:

$$247 / 1000 = 0{,}247$$
$$412 / 1000 = 0{,}412$$
$$697 / 1000 = 0{,}697$$

---

* În procesul de optimizare pot fi implicate nu numai căutările terminate cu succes, ci și cele nereușite. Căutarea cheilor care nu sunt în arbore este tot atât de costisitoare ca și căutarea celor care sunt în arbore. Pentru detalii asupra acestei probleme se poate consulta D. E. Knuth, *"Tratat de programarea calculatoarelor. Sortare și căutare"*, Secțiunea 6.2.2.

Având în vedere raportul de 5 / 2 stabilit între estimarea inițială și rezultatele instruirii, probabilitățile finale* sunt:

$$(0{,}18 \times 5 + 0{,}247 \times 2) / 7 \cong 0{,}20$$
$$(0{,}65 \times 5 + 0{,}412 \times 2) / 7 \cong 0{,}58$$
$$(0{,}17 \times 5 + 0{,}697 \times 2) / 7 \cong 0{,}22$$

Căutarea este, de fapt, o parcurgere a vârfurilor, realizată prin funcția `_search(varf<E>*, const E&)`. Această funcție nu face parte din clasa `arbore<E>`, deoarece operează exclusiv asupra vârfurilor. Iată varianta ei recursivă, împreună cu alte două funcții asemănătoare: `_min()`, pentru determinarea vârfului minim din arbore și `_succ()`, pentru determinarea succesorului†.

```
template <class E>
varf<E>* _search( varf<E>* x, const E& k ) {
  while ( x != 0 && k != x->key )
    x = k > x->key? x->dr: x->st;
  return x;
}

template <class E>
varf<E>* _min( varf<E>* x ) {
  while ( x->st != 0 )
    x = x->st;
  return x;
}

template <class E>
varf<E>* _succ( varf<E>* x ) {
  if ( x->dr != 0 ) return _min( x->dr );

  varf<E> *y = x->tata;
  while ( y != 0 && x == y->dr )
    { x = y; y = y->tata; }
  return y;
}
```

Existența acestor funcții impune completarea clasei `varf<E>` cu declarațiile `friend` corespunzătoare.

---

* Acest procedeu de estimare a probabilităților printr-un proces de instruire poate fi formalizat într-un cadru matematic riguros (R. Andonie, *"A Converse H-Theorem for Inductive Processes"*, Computers and Artificial Intelligence, Vol. 9, 1990, No. 2, pp. 159–167).

† Succesorul unui vârf *X* este vârful cu cea mai mică cheie mai mare decât cheia vârfului *X* (vezi și Exercițiul 8.10).

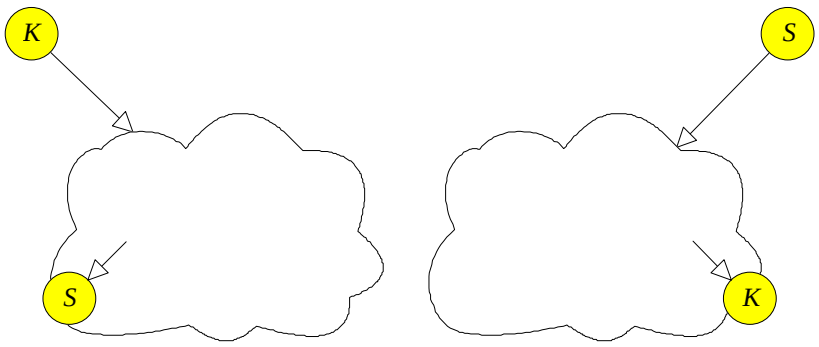
Să remarcăm asemănarea dintre funcțiile C++ de mai sus și funcțiile analoage din Exercițiul 8.10.

Pentru a demonstra corectitudinea funcțiilor `_serarch()` și `_min()`, nu avem decât să ne reamintim că, prin definiție, într-un arbore binar de căutare fiecare vârf K verifică relațiile $X \leq K$ și $K \leq Y$ pentru orice vârf X din subarborele stâng și orice vârf Y din subarborele drept.

Demonstrarea corectitudinii funcției `_succ()` este de asemenea foarte simplă. Fie K vârful al cărui succesor S trebuie determinat. Vârfurile K și S pot fi situate astfel:

- Vârful S este în subarborele drept al vârfului K. Deoarece aici sunt numai vârfuri Y cu proprietatea $K \leq Y$ (vezi Figura 8.7a) rezultă că S este valoarea minimă din acest subarbore. În plus, având în vedere procedura pentru determinarea minimului, vârful S nu are fiul stâng.
- Vârful K este în subarborele stâng al vârfului S. Deoarece fiecare vârf X de aici verifică inegalitatea $X \leq S$ (vezi Figura 8.7b), deducem că maximul din acest subarbore este chiar K. Dar maximul se determină parcurgând fiii din dreapta până la un vârf fără fiul drept. Deci, vârful K nu are fiul drept, iar S este primul ascendent din stânga al vârfului K.

În consecință, cele două situații se exclud reciproc, deci funcția `_succ()` este corectă.



(a) Vârful succesor S este minim în subarborele drept al vârfului K.

(b) Vârful K este maxim în subarborele stâng al vârfului succesor S.

Figura 8.7 Pozițiile relative ale vârfului K în raport cu sucesorul său S.

8.7.3 Modificarea arborelui

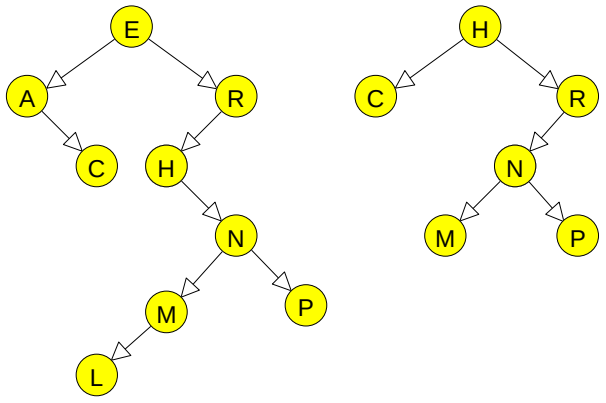
Modificarea structurii arborelui de căutare, prin inserarea sau ștergerea unor vârfuri trebuie realizată astfel încât proprietatea de arbore de căutare să nu se altereze. Cele două operații sunt diferite în privința complexității. Inserarea este simplă, fiind similară căutării. Ștergerea este mai dificilă și mult diferită de operațiile cu care deja ne-am obișnuit.

Pentru inserarea unei noi chei, vom folosi funcția

```
template <class E>
int arbore<E>::ins( const E& k, float p ) {
  varf<E> *y = 0, *x = root;

  while ( x != 0 ) {
    y = x;
    if ( k == x->key ) { // cheia deja exista in arbore
      x->p += p;         // se actualizeaza frecventa
      return 0;          // se returneaza cod de eroare
    }
    x = k > x->key? x->dr: x->st;
  }

  // cheia nu exista in arbore
  varf<E> *z = new varf<E>( k, p );
  z->tata = y;

  if ( y == 0 )                  root = z;
  else   if ( z->key > y->key ) y->dr = z;
         else                   y->st = z;

  n++;   // in arbore este cu un varf mai mult
  return 1;
}
```

Valoarea returnată este *true*, dacă cheia `k` a putut fi inserată în arbore, sau *false*, în cazul în care deja există în arbore un vârf cu cheia `k`. Inserarea propriu-zisă constă în căutarea cheii `k` prin intermediul adreselor `x` și `y`, `y` fiind adresa tatălui lui `x`. Atunci când am terminat procesul de căutare, valoarea lui `x` devine `0` și noul vârf se va insera la stânga sau la dreapta lui `y`, în funcție de relația dintre cheia `k` și cheia lui `y`.

Procedura de ștergere începe prin a determina adresa `z` a vârfului de șters, pe baza cheii `k`. Dacă procesul de căutare se finalizează cu succes, cheia `k` se va actualiza (în scopul unor prelucrări ulterioare) cu informația din vârful `z`, iar apoi se demarează procesul de ștergere efectivă a vârfului `z`. Dacă `z` este un vârf terminal, nu avem decât să anulăm legătura corespunzătoare din vârful tată. Chiar și atunci când `z` are un singur fiu, ștergerea este directă. Adresa lui `z` din vârful tată se înlocuiește cu adresa fiului lui `z`. A treia și cea mai complicată situație apare



(a) **(b)**

Figura 8.8 Ștergerea vârfurilor *E*, *A* și *L* dintr-un arbore binar de căutare.

atunci când z este situat undeva în interiorul arborelui, având ambele legături complete. În acest caz, nu vom mai șterge vârful z, ci vârful y, succesorul lui z, dar nu înainte de a copia conținutul lui y în z. Ștergerea vârfului y se face conform unuia din cele două cazuri de mai sus, deoarece, în mod sigur, y nu are fiul stâng. Într-adevăr, într-un arbore de căutare, succesorul unui vârf cu doi fii nu are fiul stâng, iar predecesorul* unui vârf cu doi fii nu are fiul drept (demonstrați acest lucru!). Pentru ilustrarea celor trei situații, am șters din arborele din Figura 8.8a vârfurile *E* (vârf cu doi fii), *A* (vârf cu un fiu) și *L* (vârf terminal).

Procedura de ștergere se implementează astfel:

```
template <class E>
int arbore<E>::del( E& k ) {
  varf<E> *z = _search( root, k );  // se cauta cheia k
  if ( !z ) return 0;               // nu a fost gasita

  n--;        // in arbore va fi cu un varf mai putin
  k = z->key; // k va retine intreaga informatie din z

  //  - y este z daca z are cel mult un fiu si
  //      succesorul lui z daca z are doi fii
  //  - x este fiul lui y sau 0 daca y nu are fii
  varf<E> *y, *x;
```

* Predecesorul unui vârf *X* este vârful care are cea mai mare cheie mai mică decât cheia vârfului *X*.

```
    y = z->st == 0 || z->dr == 0? z: _succ( z );
    x = y->st != 0?          y->st: y->dr;

    // se elimina varful y din arbore astfel:
    //  1. se stabileste legatura in x spre varful tata
    if ( x != 0 )
      x->tata = y->tata;

    //  2. in varful tata se stabileste legatura spre x
    if ( y->tata == 0 )                  root = x;
    else   if ( y == y->tata->st ) y->tata->st = x;
           else                    y->tata->dr = x;

    //  3. daca z are 2 fii, succesorul lui ii ia locul
    if ( y != z ) { z->key = y->key; z->p = y->p; }

    //  4. stergerea propriu-zisa
    y->st = y->dr = 0;
    delete y;

    return 1;
  }
```

Complexitatea funcției de ștergere este tipică pentru structurile de căutare. Aceste structuri tind să devină atât de compacte în organizarea lor internă, încât ștergerea fiecărei chei necesită reparații destul de complicate. De aceea, deseori se preferă o "ștergere leneșă" (lazy deletion), prin care vârful este doar marcat ca "șters", ștergerea efectivă realizându-se cu ocazia unor reorganizări periodice.

Deși clasa `arbore<E>` este incomplet specificată, lipsind constructorul de copiere, operatorul de atribuire, destructorul etc, operațiile implementate în această secțiune pot fi testate prin următorul program.

```
#include <iostream.h>
#include "arbore.h"

main( ) {
  int n;
  cout << "Numarul de varfuri ... "; cin >> n;

  arbore<char> g; char c; float f;

  cout << "Cheile si Frecventele lor:\n";
  for ( int i = 0; i < n; i++ ) {
    cout << "... ";
    cin >> c; cin >> f;
    g.ins( c, f );
  }
```

```
cout << "Arborele initial:\n";     g.inord( );

cout << "\n\nDelete din initial (cheie) <EOF>:\n ...";
while( cin >> c ) {
  if ( g.del( c ) ) {
    cout << "\nSe sterge varful cu cheia: " << c;
    cout << "\nInordine:\n"; g.inord( );
  }
  else
    cout << "\nelement absent";
  cout << "\n... ";
}
cin.clear( );

g.re_greedy( );
cout << "\n\nArborele Greedy:\n";  g.inord( );

cout << "\n\nInsert in Greedy "
     << "(cheie+frecventa) <EOF>:\n... ";
while( (cin >> c) && (cin >> f) ) {
  g.ins( c, f );
  cout << "\nInordine:\n"; g.inord( );
  cout << "\n... ";
}
cin.clear( );

cout << "\n\nCautari in Greedy (cheie) <EOF>:\n ...";
while( cin >> c ) {
  if ( g.search( c ) ) {
    cout << "\nNodul cu cheia: " << c;
    cout << "\nInordine:\n"; g.inord( );
  }
  else
    cout << "\nelement absent";
  cout << "\n... ";
}
cin.clear( );

cout << "\n\nDelete din Greedy (cheie) <EOF>:\n ...";
while( cin >> c ) {
  if ( g.del( c ) ) {
    cout << "\nSe sterge varful cu cheia: " << c;
    cout << "\nInordine:\n"; g.inord( );
  }
  else
    cout << "\nelement absent";
  cout << "\n... ";
}
cin.clear( );
```

```
    g.re_prodin( );
    cout << "Arborele Greedy re-ProgDin:\n"; g.inord( );

    return 1;
}
```

Funcția `arbore<E>::inord()`, definită în Secțiunea 9.2, realizează afișarea arborelui, astfel încât să poată fi ușor de reconstituit pe hârtie. De exemplu, arborele din Figura 8.8b este afișat astfel:

```
0x166c ( key C, f 0, st 0x0000, dr 0x0000, tata 0x163c )
0x163c ( key H, f 0, st 0x166c, dr 0x165c, tata 0x0000 )
0x169c ( key M, f 0, st 0x0000, dr 0x0000, tata 0x168c )
0x168c ( key N, f 0, st 0x169c, dr 0x16ac, tata 0x165c )
0x16ac ( key P, f 0, st 0x0000, dr 0x0000, tata 0x168c )
0x165c ( key R, f 0, st 0x168c, dr 0x0000, tata 0x163c )
```

8.8 Programarea dinamică comparată cu tehnica greedy

Atât programarea dinamică, cât și tehnica greedy, pot fi folosite atunci când soluția unei probleme este privită ca rezultatul unei secvențe de decizii. Deoarece principiul optimalității poate fi exploatat de ambele metode, s-ar putea să fim tentați să elaborăm o soluție prin programare dinamică, acolo unde este suficientă o soluție greedy, sau să aplicăm în mod eronat o metodă greedy, atunci când este necesară de fapt aplicarea programării dinamice. Vom considera ca exemplu o problemă clasică de optimizare.

Un hoț pătrunde într-un magazin și găsește n obiecte, un obiect i având valoarea v_i și greutatea g_i. Cum să-și optimizeze hoțul profitul, dacă poate transporta cu un rucsac cel mult o greutate G? Deosebim două cazuri. În primul dintre ele, pentru orice obiect i, se poate lua orice fracțiune $0 \le x_i \le 1$ din el, iar în al doilea caz, $x_i \in \{0,1\}$, adică orice obiect poate fi încărcat numai în întregime în rucsac. Corespunzător acestor două cazuri, obținem *problema continuă a rucsacului*, respectiv, *problema 0/1 a rucsacului*. Evident, hoțul va selecta obiectele astfel încât să maximizeze *funcția obiectiv*

$$f(x) = \sum_{i=1}^{n} v_i x_i$$

unde $x = (x_1, x_2, \ldots, x_n)$, verifică condiția

$$\sum_{i=1}^{n} g_i x_i \leq G$$

Soluția problemei rucsacului poate fi privită ca rezultatul unei secvențe de decizii. De exemplu, hoțul va decide pentru început asupra valorii lui x_1, apoi asupra valorii lui x_2 etc. Printr-o secvență optimă de decizii, el va încerca să maximizeze funcția obiectiv. Se observă că este valabil principiul optimalității. Ordinea deciziilor poate fi desigur oricare alta.

Problema continuă a rucsacului se poate rezolva prin metoda greedy, selectând la fiecare pas, pe cât posibil în întregime, obiectul pentru care v_i/g_i este maxim. Fără a restrânge generalitatea, vom presupune că

$$v_1/g_1 \geq v_2/g_2 \geq \ldots \geq v_n/g_n$$

Puteți demonstra că prin acest algoritm obținem soluția optimă și că aceasta este de forma $x^* = (1, \ldots, 1, x_k^*, 0, \ldots, 0)$, k fiind un indice, $1 \leq k \leq n$, astfel încât $0 \leq x_k \leq 1$. Algoritmul greedy găsește secvența optimă de decizii, luând la fiecare pas câte o decizie care este optimă local. Algoritmul este corect, deoarece nici o decizie din secvență nu este eronată. Dacă nu considerăm timpul necesar sortării inițiale a obiectelor, timpul este în ordinul lui n.

Să trecem la problema 0/1 a rucsacului. Se observă imediat că tehnica greedy nu conduce în general la rezultatul dorit. De exemplu, pentru $g = (1, 2, 3)$, $v = (6, 10, 12)$, $G = 5$, algoritmul greedy furnizează soluția (1, 1, 0), în timp ce soluția optimă este (0, 1, 1). Tehnica greedy nu poate fi aplicată, deoarece este generată o decizie ($x_1 = 1$) optimă local, nu însă și global. Cu alte cuvinte, la primul pas, nu avem suficientă informație locală pentru a decide asupra valorii lui x_1. Strategia greedy exploatează insuficient principiul optimalității, considerând că într-o secvență optimă de decizii fiecare decizie (și nu fiecare subsecvență de decizii, cum procedează programarea dinamică) trebuie să fie optimă. Problema se poate rezolva printr-un algoritm de programare dinamică, în această situație exploatându-se complet principiul optimalității. Spre deosebire de problema continuă, nu se cunoaște nici un algoritm polinomial pentru problema 0/1 a rucsacului.

Diferența esențială dintre tehnica greedy și programarea dinamică constă în faptul că metoda greedy generează o singură secvență de decizii, exploatând incomplet principiul optimalității. În programarea dinamică, se generează mai multe subsecvențe de decizii; ținând cont de principiul optimalității, se consideră însă doar subsecvențele optime, combinându-se acestea în soluția optimă finală. Cu toate că numărul total de secvențe de decizii este exponențial (dacă pentru fiecare din cele n decizii sunt d posibilități, atunci sunt posibile d^n secvențe de decizii), algoritmii de programare dinamică sunt de multe ori polinomiali, această reducere

a complexității datorându-se utilizării principiului optimalității. O altă caracteristică importantă a programării dinamice este că se memorează subsecvențele optime, evitându-se astfel recalcularea lor.

8.9 Exerciții

8.1 Demonstrați că numărul total de apeluri recursive necesare pentru a-l calcula pe $C(n, k)$ este $2\binom{n}{k} - 2$.

Soluție: Notăm cu $r(n, k)$ numărul de apeluri recursive necesare pentru a-l calcula pe $C(n, k)$. Procedăm prin inducție, în funcție de n. Dacă n este 0, proprietatea este adevărată. Presupunem proprietatea adevărată pentru $n-1$ și demonstrăm pentru n.

Presupunem, pentru început, că $0 < k < n$. Atunci, avem recurența

$$r(n, k) = r(n-1, k-1) + r(n-1, k) + 2$$

Din relația precedentă, obținem

$$r(n, k) = 2\binom{n-1}{k-1} - 2 + 2\binom{n-1}{k} - 2 + 2 = 2\binom{n}{k} - 2$$

Dacă k este 0 sau n, atunci $r(n, k) = 0$ și, deoarece în acest caz avem $\binom{n}{k} = 1$, rezultă că proprietatea este adevărată. Acest rezultat poate fi verificat practic, rulând programul din Exercițiul 2.5.

8.2 Arătați că principiul optimalității

i) este valabil în problema găsirii celui mai scurt drum dintre două vârfuri ale unui graf

ii) nu este valabil în problema determinării celui mai lung drum simplu dintre două vârfuri ale unui graf

8.3 Demonstrați că $\binom{2n}{k} \geq 4^n/(2n+1)$.

8.4 Folosind algoritmul *serie*, calculați probabilitatea ca jucătorul A să câștige, presupunând $n = 4$ și $p = 0{,}45$.

8.5 Problema înmulțirii înlănțuite optime a matricilor se poate rezolva și prin următorul algoritm recursiv:

```
function rminscal(i, j)
    {returnează numărul minim de înmulțiri scalare
     pentru a calcula produsul matricial M_i M_{i+1} ... M_j}
    if i = j then return 0
    q ← +∞
    for k ← i to j–1 do
        q ← min(q, rminscal(i, k)+rminscal(k+1, j)+d[i–1]d[k]d[ j])
    return q
```

unde tabloul $d[0 .. n]$ este global. Găsiți o limită inferioară a timpului. Explicați ineficiența acestui algoritm.

Soluție: Notăm cu $r(j–i+1)$ numărul de apeluri recursive necesare pentru a-l calcula pe *rminscal*(i, j). Pentru $n > 2$ avem

$$r(n) = \sum_{k=1}^{n-1} r(k) + r(n-k) = 2\sum_{k=1}^{n-1} r(k) \geq 2r(n-1)$$

iar $r(2) = 2$. Prin metoda iterației, deduceți că $r(n) \geq 2^{n-1}$, pentru $n > 2$. Timpul pentru un apel *rminscal*$(1, n)$ este atunci în $\Omega(2^n)$.

8.6 Elaborați un algoritm eficient care să afișeze parantezarea optimă a unui produs matricial $M(1), \ldots, M(n)$. Folosiți pentru aceasta matricea r, calculată de algoritmul *minscal*. Analizați algoritmul obținut.

Soluție: Se apelează cu *paran*$(1, n)$ următorul algoritm:

```
function paran(i, j)
    if i = j  then  write "M(", i, ")"
              else  write "("
                    parant(i, r[i, j])
                    write "*"
                    parant(r[i, j]+1, j)
                    write ")"
```

Arătați prin inducție că o parantezare completă unei expresii de n elemente are exact $n–1$ perechi de paranteze. Deduceți de aici care este eficiența algoritmului.

8.7 Presupunând matricea P din algoritmul lui Floyd cunoscută, elaborați un algoritm care să afișeze prin ce vârfuri trece cel mai scurt drum dintre două vârfuri oarecare.

8.8 Într-un graf orientat, să presupunem că ne interesează doar existența, nu și lungimea drumurilor, între fiecare pereche de vârfuri. Inițial, $L[i, j]$ = *true* dacă muchia (i, j) există și $L[i, j]$ = *false* în caz contrar. Modificați algoritmul lui Floyd astfel încât, în final, să avem $D[i, j]$ = *true* dacă există cel puțin un drum de la i la j și $D[i, j]$ = *false* în caz contrar.

Soluție: Se înlocuiește bucla cea mai interioară cu:

$$D[i, j] \leftarrow D[i, j] \textbf{ or } (D[i, k] \textbf{ and } D[k, j])$$

obținându-se algoritmul lui Warshall (1962). Matricea booleană L se numește *închiderea tranzitivă* a grafului.

8.9 Arătați cu ajutorul unui contraexemplu că următoarea propoziție nu este, în general, adevărată: "Un arbore binar este un arbore de căutare dacă cheia fiecărui vârf neterminal este mai mare sau egală cu cheia fiului său stâng și mai mică sau egală cu cheia fiului său drept".

8.10 Fie un arbore binar de căutare reprezentat prin adrese, astfel încât vârful i (adică vârful a cărui adresă este i) este memorat în patru locații diferite conținând :

$KEY[i]$	=	cheia vârfului
$ST[i]$	=	adresa fiului stâng
$DR[i]$	=	adresa fiului drept
$TATA[i]$	=	adresa tatălui

(Dacă se folosește o implementare prin tablouri paralele, atunci adresele sunt indici de tablou). Presupunem că variabila *root* conține adresa rădăcinii arborelui și că o adresă este zero, dacă și numai dacă vârful către care se face trimiterea lipsește. Elaborați algoritmi pentru următoarele operații în arborele de căutare:

i) Determinarea vârfului care conține o cheie v dată. Dacă un astfel de vârf nu există, se va returna adresa zero.

ii) Determinarea vârfului care conține cheia minimă.

iii) Determinarea succesorului unui vârf i dat (*succesorul* vârfului i este vârful care are cea mai mică cheie mai mare decât $KEY[i]$).

Care este eficiența acestor algoritmi?

Soluție:

i) Apelăm *tree-search*(*root*, *v*), *tree-search* fiind funcția:

```
function tree-search(i, v)
    if i = 0 or v = KEY[i] then return i
    if v < KEY[i]   then  return tree-search(ST[i], v)
                    else  return tree-search(DR[i], v)
```

Iată și o versiune iterativă a acestui algoritm:

```
function iter-tree-search(i, v)
    while i ≠ 0 and v ≠ KEY[i] do
        if i < KEY[i]then  i ← ST[i]
                     else  i ← DR[i]
    return i
```

ii) Se apelează *tree-min*(*root*), *tree-min* fiind funcția:

```
function tree-min(i)
    while ST[i] ≠ 0 do i ← ST[i]
    return i
```

iii) Următorul algoritm returnează succesorul vârfului *i*:

```
function tree-succesor(i)
    if DR[i] ≠ 0 then return tree-min(DR[i])
    j ← TATA[i]
    while j ≠ 0 and i = DR[ j]  do  i ← j
                                    j ← TATA[ j]
    return j
```

8.11 Găsiți o formulă explicită pentru $T(n)$, unde $T(n)$ este numărul de arbori de căutare diferiți care se pot construi cu n chei distincte.

Indicație: Faceți legătura cu problema înmulțirii înlănțuite a matricilor.

8.12 Există un algoritm greedy evident pentru a construi arborele optim de căutare având cheile $c_1 < c_2 < \ldots < c_n$: se plasează cheia cea mai probabilă, c_k, la rădăcină și se construiesc subarborii săi stâng și drept pentru cheile $c_1, c_2, \ldots, c_{k-1}$, respectiv, $c_{k+1}, c_{k+2}, \ldots, c_n$, în mod recursiv, pe același principiu.

i) Cât timp necesită algoritmul pentru cazul cel mai nefavorabil?

ii) Arătați pe baza unui contraexemplu că prin acest algoritm greedy nu se obține întotdeauna arborele optim de căutare.

8.13 Un subcaz oarecare al problemei 0/1 a rucsacului se poate formula astfel:

Să se găsească

$$V(l, j, X) = \max \sum_{l \le i \le j} v_i x_i$$

unde maximul se ia pentru toți vectorii $(x_l, \ldots, x_j)$ pentru care

$$\sum_{l \le i \le j} g_i x_i \le X$$

$$x_i \in \{0, 1\}, \quad l \le i \le j$$

În particular, $V(1, n, G)$ este valoarea maximă care se poate încărca în rucsac în cazul problemei inițiale. O soluție a acestei probleme se poate obține dacă considerăm că deciziile se iau *retrospectiv*, adică în ordinea $x_n, x_{n-1}, \ldots, x_1$. Principiul optimalității este valabil și avem

$$V(1, n, G) = \max(V(1, n-1, G), V(1, n-1, G-g_n) + v_n)$$

și, în general,

$$V(1, j, X) = \max(V(1, j-1, X), V(1, j-1, X-g_j) + v_j)$$

unde $V(1, 0, X) = 0$ pentru $X \ge 0$, iar $V(1, j, X) = -\infty$ pentru $X < 0$. De aici se poate calcula, prin tehnica programării dinamice, valoarea $V(1, n, G)$ care ne interesează.

Găsiți o recurență similară pentru situația când deciziile se iau *prospectiv*, adică în ordinea $x_1, x_2, \ldots, x_n$.

8.14 Am văzut (în Secțiunea 6.1) că tehnica greedy poate fi aplicată în problema determinării restului cu un număr minim de monezi doar pentru anumite cazuri particulare. Problema se poate rezolva, în cazul general, prin metoda programării dinamice.

Să presupunem că avem un număr finit de n tipuri de monezi, fiecare în număr nelimitat, iar tabloul $M[1 .. n]$ conține valoarea acestor monezi. Fie S suma pe care dorim să o obținem, folosind un număr minim de monezi.

i) În tabloul $C[1 .. n, 1 .. S]$, fie $C[i, j]$ numărul minim de monezi necesare pentru a obține suma j, folosind doar monezi de tipul $M[1], M[2], \ldots, M[i]$, unde $C[i, j] = +\infty$, dacă suma j nu poate fi obținută astfel. Găsiți o recurență pentru $C[i, j]$.

ii) Elaborați un algoritm care folosește tehnica programării dinamice pentru a calcula valorile $C[n, j]$, $1 \le j \le S$. Algoritmul trebuie să utilizeze un singur vector de S elemente. Care este timpul necesar, în funcție de n și S?

iii) Găsiți un algoritm greedy care determină cum se obține suma S cu un număr minim de monezi, presupunând cunoscute valorile $C[n, j]$.

8.15 Fie u și v două secvențe de caractere. Dorim să transformăm pe u în v, cu un număr minim de operații de următoarele tipuri:

- șterge un caracter
- adaugă un caracter
- schimbă un caracter

De exemplu, putem să transformăm *abbac* în *abcbc* în trei etape:

$$\begin{aligned} abbac &\rightarrow abac && \text{(șterge } b\text{)} \\ &\rightarrow ababc && \text{(adaugă } b\text{)} \\ &\rightarrow abcbc && \text{(schimbă } a \text{ cu } c\text{)} \end{aligned}$$

Arătați că această transformare nu este optimă. Elaborați un algoritm de programare dinamică care găsește numărul minim de operații necesare (și le specifică) pentru a-l transforma pe u în v.

8.16 Să considerăm alfabetul $\Sigma = \{a, b, c\}$. Pentru elementele lui Σ definim următoarea tablă de înmulțire:

		simbolul drept		
		a	*b*	*c*
simbolul	*a*	*b*	*b*	*a*
stâng	*b*	*c*	*b*	*a*
	c	*a*	*c*	*c*

Observați că înmulțirea definită astfel nu este nici comutativă și nici asociativă. Găsiți un algoritm eficient care examinează șirul $x = x_1\, x_2 \ldots x_n$ de caractere ale lui Σ și decide dacă x poate fi parantezat astfel încât expresia rezultată să fie a. De exemplu, dacă $x = bbbba$, algoritmul trebuie să returneze "da" deoarece $(b(bb))(ba) = a$.

8.17 Arătați că numărul de moduri în care un poligon convex cu n laturi poate fi partiționat în $n-2$ triunghiuri, folosind linii diagonale care nu se întretaie, este $T(n-1)$, unde $T(n-1)$ este al $(n-1)$-lea număr catalan.

9. Explorări în grafuri

Am văzut deja că o mare varietate de probleme se formulează în termeni de grafuri. Pentru a le rezolva, de multe ori trebuie să *explorăm* un graf, adică să consultăm (vizităm) vârfurile sau muchiile grafului respectiv. Uneori trebuie să consultăm toate vârfurile sau muchiile, alteori trebuie să consultăm doar o parte din ele. Am presupus, până acum, că există o anumită ordine a acestor consultări: cel mai apropiat vârf, cea mai scurtă muchie etc. În acest capitol, introducem câteva tehnici care pot fi folosite atunci când nu este specificată o anumită ordine a consultărilor.

Vom folosi termenul de "graf" în două ipostaze. Un graf va fi uneori, ca şi până acum, o structură de date implementată în memoria calculatorului. Acest mod *explicit* de reprezentare nu este însă indicat atunci când graful conține foarte multe vârfuri.

Să presupunem, de exemplu, că folosim vârfurile unui graf pentru a reprezenta configurații în jocul de șah, fiecare muchie corespunzând unei mutări legale între două configurații. Acest graf are aproximativ 10^{120}vârfuri. Presupunând că un calculator ar fi capabil să genereze 10^{11} vârfuri pe secundă, generarea completă a grafului asociat jocului de șah s-ar face în mai mult de 10^{80} ani! Un graf atât de mare nu poate să aibă decât o existență implicită, abstractă.

Un graf *implicit* este un graf reprezentat printr-o descriere a vârfurilor şi muchiilor sale, el neexistând integral în memoria calculatorului. Porțiuni relevante ale grafului pot fi construite pe măsură ce explorarea progresează. De exemplu, putem avea în memorie doar o reprezentare a vârfului curent şi a muchiilor adiacente lui; pe măsură ce înaintăm în graf, vom actualiza această reprezentare.

Tehnicile de explorare pentru cele două concepte de graf (grafuri construite explicit şi grafuri implicite) sunt, în esență, identice. Indiferent de obiectivul urmărit, explorarea se realizează pe baza unor *algoritmi de parcurgere*, care asigură consultarea sistematică a vârfurilor sau muchiilor grafului respectiv.

9.1 Parcurgerea arborilor

Pentru parcurgerea arborilor binari există trei tehnici de bază. Dacă pentru fiecare vârf din arbore vizităm prima dată vârful respectiv, apoi vârfurile din subarborele stâng şi, în final, subarborele drept, înseamnă că parcurgem arborele în *preordine*. Dacă vizităm subarborele stâng, vârful respectiv şi apoi subarborele drept, atunci

parcurgem arborele în *inordine*, iar dacă vizităm prima dată subarborele stâng, apoi cel drept, apoi vârful respectiv, parcurgerea este în *postordine*. Toate aceste tehnici parcurg arborele de la stânga spre dreapta. Putem parcurge însă arborele și de la dreapta spre stânga, obținând astfel încă trei moduri de parcurgere.

Proprietatea 9.1 Pentru fiecare din aceste șase tehnici de parcurgere, timpul necesar pentru a explora un arbore binar cu n vârfuri este în $\Theta(n)$.

Demonstrație: Fie $t(n)$ timpul necesar pentru parcurgerea unui arbore binar cu n vârfuri. Putem presupune că există constanta reală pozitivă c, astfel încât $t(n) \leq c$ pentru $0 \leq n \leq 1$. Timpul necesar pentru parcurgerea unui arbore cu n vârfuri, $n > 1$, în care un vârf este rădăcina, i vârfuri sunt situate în subarborele stâng și $n-i-1$ vârfuri în subarborele drept, este

$$t(n) \leq c + \max \{t(i)+t(n-i-1) \mid 0 \leq i \leq n-1\}$$

Vom arăta, prin inducție constructivă, că $t(n) \leq dn+c$, unde d este o altă constantă. Pentru $n = 0$, proprietatea este adevărată. Prin ipoteza inducției specificate parțial, presupunem că $t(i) \leq di+c$, pentru orice $0 \leq i < n$. Demonstrăm că proprietatea este adevărată și pentru n. Avem

$$t(n) \leq c+2c+d(n-1) = dn+c+2c-d$$

Luând $d \geq 2c$, obținem $t(n) \leq dn+c$. Deci, pentru d suficient de mare, $t(n) \leq dn+c$, pentru orice $n \geq 0$, adică $t \in O(n)$. Pe de altă parte, $t \in \Omega(n)$, deoarece fiecare din cele n vârfuri trebuie vizitat. În consecință, $t \in \Theta(n)$. ■

Pentru fiecare din aceste tehnici de parcurgere, implementarea recursivă necesită, în cazul cel mai nefavorabil, un spațiu de memorie în $\Omega(n)$ (demonstrați acest lucru!). Cu puțin efort*, tehnicile menționate pot fi implementate astfel încât să necesite un timp în $\Theta(n)$ și un spațiu de memorie în $\Theta(1)$, chiar dacă vârfurile nu conțin adresa tatălui (caz în care problema devine trivială).

Conceptele de preordine și postordine se pot generaliza pentru arbori arbitrari (nebinari). Timpul de parcurgere este tot în ordinul numărului de vârfuri.

---

* O astfel de implementare poate fi găsită, de exemplu, în E. Horowitz și S. Sahni, *"Fundamentals of Computer Algorithms"*, Secțiunea 6.1.1.

9.2 Operații de parcurgere în clasa arbore<E>

Tipul abstract arbore este imposibil de conceput în lipsa unor metode sistematice de explorare. Iată câteva situații în care le-am folosit, sau va trebui să le folosim:

- Reorganizarea într-un arbore de căutare optim. Este vorba de procedura `setvarf()` din clasa `s8a` (Secțiunea 8.7.1), procedură prin care s-a inițializat un tablou cu adresele tuturor vârfurilor din arbore. Acum este clar că am folosit o parcurgere în inordine, prilej cu care am ajuns și la o procedură de sortare similară *quicksort*-ului.
- Copierea, vârf cu vârf, a unui arbore într-un alt arbore. Procedura este necesară constructorului și operatorului de atribuire.
- Implementarea destructorului clasei, adică eliberarea spațiului ocupat de fiecare din vârfurile arborelui.
- Afișarea unor "instantanee" ale structurii arborilor pentru a verifica corectitudinea diverselor operații.

Operația de copiere este implementată prin funcția `_copy()` din clasa `varf<E>`. Este vorba de o funcție care copiază recursiv arborele al cărui vârf rădăcină este dat ca argument, iar apoi returnează adresa arborelui construit prin copiere.

```
template <class E>
varf<E>* _copy( varf<E>* x ) {
  varf<E> *z = 0;
  if ( x ) {
    // varful nou alocat se initializeaza cu x
    z = new varf<E>( x->key, x->p );

    // se copiaza subarborii din stanga si din deapta; in
    // fiecare se initializeaza legatura spre varful tata
    if ( (z->st = _copy( x->st )) != 0 )
      z->st->tata = z;
    if ( (z->dr = _copy( x->dr )) != 0 )
      z->dr->tata = z;
  }
  return z;
}
```

Invocarea acestei funcții este realizată atât de către constructorul de copiere al clasei arbore,

```
template <class E>
arbore<E>::arbore( const arbore<E>& a ) {
  root = _copy( a.root ); n = a.n;
}
```

cât şi de către operatorul de atribuire:

```
template <class E>
arbore<E>& arbore<E>::operator =( const arbore<E>& a ) {
  delete root;
  root = _copy( a.root ); n = a.n;
  return *this;
}
```

Efectul instrucţiunii `delete root` ar trebui să fie ştergerea tuturor vârfurilor din arborele cu rădăcina `root`. Pentru a ajunge la acest rezultat, avem nevoie de implementarea corespunzătoare a destructorului clasei `varf<E>`, destructor invocat, după cum se ştie, înainte ca operatorul `delete` să elibereze spaţiul alocat. Forma acestui destructor este foarte simplă:

```
~varf( ) { delete st; delete dr; }
```

Efectul lui constă în ştergerea vârfurilor în postordine. Mai întâi, se acţionează asupra sub-arborelui stâng, apoi asupra celui drept, iar în final, după execuţia corpului destructorului, operatorul `delete` eliberează spaţiul alocat vârfului curent. Condiţia de oprire a recursivităţii este asigurată de operatorul `delete`, el fiind inefectiv pentru adresele nule. În consecinţă, şi destructorul clasei `arbore<E>` constă într-un simplu `delete root`:

```
~arbore( ) { delete root; }
```

Toate modalităţile de parcurgere menţionate în Secţiunea 9.1 pot fi implementate imediat, prin funcţiile corespunzătoare. Noi ne-am rezumat la implementarea parcurgerii în inordine deoarece, pe parcursul testării clasei `arbore<E>`, am avut nevoie de afişarea structurii arborelui. Funcţia

```
template <class E>
void _inord( varf<E> *x ) {

  if ( !x ) return;

  _inord( x->st );
```

```
    cout << x
         << " ( key " << x->key
         << ", f "    << x->p
         << ", st "   << x->st
         << ", dr "   << x->dr
         << ", tata " << x->tata
         << " )";

    _inord( x->dr );
  }
```

apelabilă din clasa `arbore<E>` prin

```
template <class E>
void arbore<E>::inord( ) { _inord( root ); }
```

este exact ceea ce ne trebuie pentru a afișa întreaga structură internă a arborelui.

9.3 Parcurgerea grafurilor în adâncime

Fie $G = <V, M>$ un graf orientat sau neorientat, ale cărui vârfuri dorim să le consultăm. Presupunem că avem posibilitatea să marcăm vârfurile deja vizitate în tabloul global *marca*. Inițial, nici un vârf nu este marcat.

Pentru a efectua o parcurgere *în adâncime*, alegem un vârf oarecare, $v \in V$, ca punct de plecare și îl marcăm. Dacă există un vârf w adiacent lui v (adică, dacă există muchia (v, w) în graful orientat G, sau muchia $\{v, w\}$ în graful neorientat G) care nu a fost vizitat, alegem vârful w ca noul punct de plecare și apelăm recursiv procedura de parcurgere în adâncime. La întoarcerea din apelul recursiv, dacă există un alt vârf adiacent lui v care nu a fost vizitat, apelăm din nou procedura etc. Când toate vârfurile adiacente lui v au fost marcate, se încheie consultarea începută în v. Dacă au rămas vârfuri în V care nu au fost vizitate, alegem unul din aceste vârfuri și apelăm procedura de parurgere. Continuăm astfel, până când toate vârfurile din V au fost marcate. Iată algoritmul:

procedure *parcurge*(G)
  for fiecare $v \in V$ **do** *marca*[v] ← *nevizitat*
  for fiecare $v \in V$ **do**
    if *marca*[v] = *nevizitat* **then** *ad*(v)

procedure *ad*(v)
  {vîrful v nu a fost vizitat}
  marca[v] ← *vizitat*
  for fiecare vîrf w adiacent lui v **do**
    if *marca*[w] = *nevizitat* **then** *ad*(w)

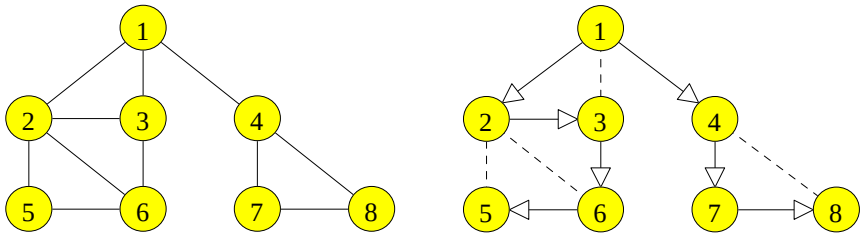


Figura 9.1 Un graf neorientat și unul din arborii săi parțiali.

Acest mod de parcurgere se numește "în adâncime", deoarece încearcă să inițieze cât mai multe apeluri recursive înainte de a se întoarce dintr-un apel.

Parcurgerea în adâncime a fost formulată cu mult timp în urmă ca o tehnică de explorare a unui labirint. O persoană care caută ceva într-un labirint și aplică această tehnică are avantajul că "următorul loc în care caută" este mereu foarte aproape.

Pentru graful din Figura 9.1a, presupunând că pornim din vârful 1 și că vizităm vecinii unui vârf în ordine numerică, parcurgerea vârfurilor în adâncime se face în ordinea: 1, 2, 3, 6, 5, 4, 7, 8.

Desigur, parcurgerea în adâncime a unui graf nu este unică; ea depinde atât de alegerea vârfului inițial, cât și de ordinea de vizitare a vârfurilor adiacente.

Cât timp este necesar pentru a parcurge un graf cu n vârfuri și m muchii? Deoarece fiecare vârf este vizitat exact o dată, avem n apeluri ale procedurii *ad*. În procedura *ad*, când vizităm un vârf, testăm marcajul fiecărui vecin al său. Dacă reprezentăm graful prin liste de adiacență, adică prin atașarea la fiecare vârf a listei de vârfuri adiacente lui, atunci numărul total al acestor testări este: m, dacă graful este orientat, și $2m$, dacă graful este neorientat. Algoritmul necesită un timp în $\Theta(n)$ pentru apelurile procedurii *ad* și un timp în $\Theta(m)$ pentru inspectarea mărcilor. Timpul de execuție este deci în $\Theta(\max(m, n)) = \Theta(m+n)$.

Dacă reprezentăm graful printr-o matrice de adiacență, se obține un timp de execuție în $\Theta(n^2)$.

Parcurgerea în adâncime a unui graf G, neorientat și conex, asociază lui G un arbore parțial. Muchiile arborelui corespund muchiilor parcurse în G, iar vârful ales ca punct de plecare devine rădăcina arborelui. Pentru graful din Figura 9.1a, un astfel de arbore este reprezentat în Figura 9.1b prin muchiile "continue"; muchiile din G care nu corespund unor muchii ale arborelui sunt "punctate". Dacă

graful *G* nu este conex, atunci parcurgerea în adâncime asociază lui *G* o pădure de arbori, câte unul pentru fiecare componentă conexă a lui *G*.

Dacă dorim să și marcăm numeric vârfurile în ordinea parcurgerii lor, adăugăm în procedura *ad*, la început:

$num \leftarrow num + 1$
$preord[v] \leftarrow num$

unde *num* este o variabilă globală inițializată cu zero, iar $preord[1 .. n]$ este un tablou care va conține în final ordinea de parcurgere a vârfurilor. Pentru parcurgerea din exemplul precedent, acest tablou devine:

1	2	3	6	5	4	7	8

Cu alte cuvinte, se parcurg în preordine vârfurile arborelui parțial din Figura 9.1b.

Se poate observa că parcurgerea în adâncime a unui arbore, pornind din rădăcină, are ca efect parcurgerea în preordine a arborelui.

9.3.1 Puncte de articulare

Parcurgerea în adâncime se dovedește utilă în numeroase probleme din teoria grafurilor, cum ar fi: detectarea componentelor conexe (respectiv, tare conexe) ale unui graf, sau verificarea faptului că un graf este aciclic. Ca exemplu, vom rezolva în această secțiune problema găsirii punctelor de articulare ale unui graf conex.

Un vârf *v* al unui graf neorientat conex este un *punct de articulare*, dacă subgraful obținut prin eliminarea lui *v* și a muchiilor care plecă din *v* nu mai este conex. De exemplu, vârful 1 este un punct de articulare pentru graful din Figura 9.1. Un graf neorientat este *biconex* (sau *nearticulat*) dacă este conex și nu are puncte de articulare. Grafurile biconexe au importante aplicații practice: dacă o rețea de telecomunicații poate fi reprezentată printr-un graf biconex, aceasta ne garantează că rețeaua continuă să funcționeze chiar și după ce echipamentul dintr-un vârf s-a defectat.

Este foarte util să putem verifica eficient dacă un graf are puncte de articulare. Următorul algoritm găsește punctele de articulare ale unui graf conex *G*.

1. Efectuează o parcurgere în adâncime a lui *G* pornind dintr-un vârf oarecare. Fie *A* arborele parțial generat de această parcurgere și *preord* tabloul care conține ordinea de parcurgere a vârfurilor.

2. Parcurge arborele *A* în postordine. Pentru fiecare vârf *v* vizitat, calculează $minim[v]$ ca minimul dintre

- *preord*[*v*]
- *preord*[*w*] pentru fiecare vârf *w* pentru care există o muchie {*v*, *w*} în *G* care nu are o muchie corespunzătoare în *A* (în Figura 9.1b, o muchie "punctată")
- *minim*[*x*] pentru fiecare fiu *x* al lui *v* în *A*

3. Punctele de articulare se determină acum astfel:

a. rădăcina lui *A* este un punct de articulare al lui *G*, dacă și numai dacă are mai mult de un fiu;

b. un vârf *v* diferit de rădăcina lui *A* este un punct de articulare al lui *G*, dacă și numai dacă *v* are un fiu *x*, astfel încât *minim*[*x*] ≥ *preord*[*v*].

Pentru exemplul din Figura 9.1b, rezultă că tabloul *minim* este

1	1	1	6	2	2	6	6

iar vârfurile 1 și 4 sunt puncte de articulare.

Pentru a demonstra că algoritmul este corect, enunțăm pentru început o proprietate care rezultă din Exercițiul 9.8: orice muchie din *G*, care nu are o muchie corespunzătoare în *A*, conectează în mod necesar un vârf *v* cu un ascendent al său în *A*. Ținând cont de această proprietate, valoarea *minim*[*v*] se poate defini și astfel:

minim[*v*] = min{*preord*[*w*] | se poate ajunge din *v* în *w* urmând oricâte muchii "continue", iar apoi urmând "în sus" cel mult o muchie "punctată"}

Alternativa **3a** din algoritm rezultă imediat, deoarece este evident că rădăcina lui *A* este un punct de articulare al lui *G*, dacă și numai dacă are mai mult de un fiu.

Să presupunem acum că *v* nu este rădăcina lui *A*. Dacă *x* este un fiu al lui *v* și *minim*[*x*] < *preord*[*v*], rezultă că există o succesiune de muchii care îl conectează pe *x* cu celelalte vârfuri ale grafului, chiar și după eliminarea lui *v*. Pe de altă parte, nu există nici o succesiune de muchii care să îl conecteze pe *x* cu tatăl lui *v*, dacă *minim*[*x*] ≥ *preord*[*v*]. Se deduce că și alternativa **3b** este corectă.

9.3.2 Sortarea topologică

În această secțiune, vom arăta cum putem aplica parcurgerea în adâncime a unui graf, într-un procedeu de sortare esențial diferit față de sortările întâlnite până acum.

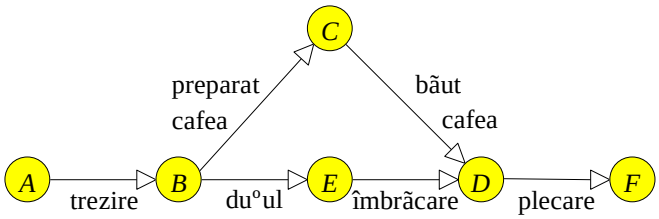


Figura 9.2 Un graf orientat aciclic.

Să presupunem că reprezentăm diferitele stagii ale unui proiect complex printr-un graf orientat aciclic: vârfurile sunt stările posibile ale proiectului, iar muchiile corespund activităților care se cer efectuate pentru a trece de la o stare la alta. Figura 9.2 dă un exemplu al acestui mod de reprezentare. O sortare topologică a vârfurilor unui graf orientat aciclic este o operație de ordonare liniară a vârfurilor, astfel încât, dacă există o muchie (i, j), atunci i apare înaintea lui j în această ordonare.

Pentru graful din Figura 9.2, o sortare topologică este *A*, *B*, *C*, *E*, *D*, *F*, iar o alta este *A*, *B*, *E*, *C*, *D*, *F*. În schimb, secvența *A*, *B*, *C*, *D*, *E*, *F* nu este în ordine topologică.

Dacă adăugăm la sfîrșitul procedurii *ad* linia

write *v*

atunci procedura de parcurgere în adâncime va afișa vârfurile în ordine topologică inversă. Pentru a înțelege de ce se întîmplă acest lucru, să observăm că vârful *v* este afișat *după ce* toate vârfurile către care există o muchie din *v* au fost deja afișate.

9.4 Parcurgerea grafurilor în lățime

Procedura de parcurgere în adâncime, atunci când se ajunge la un vârf *v* oarecare, explorează prima dată un vârf *w* adiacent lui *v*, apoi un vârf adiacent lui *w* etc. Pentru a efectua o parcurgere *în lățime* a unui graf (orientat sau neorientat), aplicăm următorul principiu: atunci când ajungem într-un vârf oarecare *v* nevizitat, îl marcăm și vizităm apoi toate vârfurile nevizitate adiacente lui *v*, apoi toate vârfurile nevizitate adiacente vârfurilor adiacente lui *v* etc. Spre deosebire de parcurgerea în adâncime, parcurgerea în lățime nu este în mod natural recursivă.

Pentru a putea compara aceste două tehnici de parcurgere, vom da pentru început o versiune nerecursivă pentru procedura *ad*. Versiunea se bazează pe utilizarea unei stive. Presupunem că avem funcția *ftop* care returnează ultimul vârf inserat în stivă, fără să îl șteargă. Folosim și funcțiile *push* și *pop* din Secțiunea 3.1.1.

```
procedure iterad(v)
    S ← stivă vidă
    marca[v] ← vizitat
    push(v, S)
    while S nu este vidă do
        while există un vârf w adiacent lui ftop(S)
                astfel încât marca[w] = nevizitat   do
            marca[w] ← vizitat
            push(w, S)
        pop(S)
```

Pentru parcurgerea în lățime, vom utiliza o coadă și funcțiile *insert-queue, delete-queue* din Secțiunea 3.1.2. Iată acum algoritmul de parcurgere în lățime:

```
procedure lat(v)
    C ← coadă vidă
    marca[v] ← vizitat
    insert-queue(v, C)
    while C nu este vidă do
        u ← delete-queue(C)
        for fiecare vârf w adiacent lui u do
            if marca[w] = nevizitat  then   marca[w] ← vizitat
                                            insert-queue(w, C)
```

Procedurile *iterad* și *lat* trebuie apelate din procedura

```
procedure parcurge(G)
    for fiecare v ∈ V do marca[v] ← nevizitat
    for fiecare v ∈ V do
        if marca[v] = nevizitat then {iterad sau lat} (v)
```

De exemplu, pentru graful din Figura 9.1, ordinea de parcurgere în lățime a vârfurilor este: 1, 2, 3, 4, 5, 6, 7, 8.

Ca și în cazul parcurgerii în adâncime, parcurgerea în lățime a unui graf G conex asociază lui G un arbore parțial. Dacă G nu este conex, atunci obținem o pădure de arbori, câte unul pentru fiecare componentă conexă.

Analiza eficienței algoritmului de parcurgere în lățime se face la fel ca pentru parcurgerea în adâncime. Pentru a parcurge un graf cu n vârfuri și m muchii

timpul este în: *i)* $\Theta(n+m)$, dacă reprezentăm graful prin liste de adiacență; *ii)* $\Theta(n^2)$, dacă reprezentăm graful printr-o matrice de adiacență.

Parcurgerea în lățime este folosită de obicei atunci când se explorează parțial anumite grafuri infinite, sau când se caută cel mai scurt drum dintre două vârfuri.

9.5 Salvarea și restaurarea arborilor binari de căutare

Importanța operațiilor de *salvare* (*backup*) și *restaurare* (*restore*) este bine cunoscută de către toți utilizatorii de calculatoare. Într-un fel sau altul, este bine ca informațiile să fie arhivate periodic pe un suport extern, astfel ca, în caz de necesitate, să le putem reconstitui cât mai ușor. Pentru clasa `arbore<E>` am decis să implementăm operațiile de salvare și restaurare, în scopul de a facilita transferurile de arbori între programe. Vom exemplifica cu această ocazie, nu numai parcurgerea în lățime, ci și lucrul cu fișiere binare, prin intermediul obiectelor de tip `fstream` din biblioteca standard de intrare/ieșire a limbajului C++, obiecte declarate în fișierul header `<fstream.h>`.

Convenim să memorăm pe suportul extern atât cheia, cât și probabilitatea (frecvența) de acces a fiecărui vârf. Scrierea se va face cheie după cheie (vârf după vârf), în ordinea obținută printr-un proces de vizitare a arborelui. Restaurarea arborelui este realizată prin inserarea fiecărei chei într-un arbore inițial vid. Citirea cheilor este secvențială, adică în ordinea în care au fost scrise în fișier.

Parcurgerile în adâncime (în preordine) și în lățime au proprietatea că vârful rădăcină al arborelui și al fiecărui subarbore este vizitat (și deci inserat) înaintea vârfurilor fii. Avem astfel garantată reconstituirea corectă a arborelui de căutare, deoarece în momentul în care se inserează o cheie oarecare, toate vârfurile ascendente sunt deja inserate. În cele ce urmează, vom utiliza parcurgerea în lățime.

Parcurgerea în lățime a arborilor binari se face conform algoritmului din Secțiunea 9.4, cu specificarea că, deoarece arborii sunt grafuri conexe și aciclice, nu mai este necesară marcarea vârfurilor. În procedura de salvare,

```
template <class E>
int arbore<E>::save( char *file ) {
  ofstream f( file, ios::binary );  // deschide fisierul
  if ( !f ) return 0;               // eroare la deschidere

  coada<varf<E>*> c( n + 1 ); // ptr. parcurgerea in latime
  varf<E> *x;                 // varful curent

  c.ins_q( root );            // primul element din coada
  while ( c.del_q( x ) ) {
    if ( !f.write( (char *) &(x->key), sizeof( x->key ) ) )
      return 0;               // eroare la scriere
    if ( !f.write( (char *) &(x->p  ), sizeof( x->p   ) ) )
      return 0;               // eroare la scriere

    if ( x->st ) c.ins_q( x->st );
    if ( x->dr ) c.ins_q( x->dr );
  }
  f.close( );
  return 1;
}
```

vizitarea unui vârf constă în scrierea informațiilor asociate în fișierul de ieșire. De această dată, nu vom mai folosi operatorii de ieșire `>>` ai claselor `E` și `float`, ci vom copia, octet cu octet, imaginea binară a cheii și a probabilității asociate. Cheia este situată la adresa `&(x->key)` și are lungimea `sizeof(x->key)`, sau `sizeof(E)`. Probabilitatea este situată la adresa `&(x->p)` și are lungimea `sizeof(x->p)`, sau `sizeof(float)`. Operația de scriere necesită un obiect de tip `ofstream`, *output file stream*, creat pe baza numelui fișierului `char *file`. Prin valoarea `ios::binary` din lista de argumente a constructorului clasei `ofstream`, fișierul va fi deschis în modul binar de lucru și nu în modul implicit text.

Funcția de restaurare

```
template <class E>
int arbore<E>::rest( char *file ) {
  ifstream f( file, ios::binary ); // deschide fisierul
  if ( !f ) return 0;              // eroare la deschidere

  delete root;
  root = 0; n = 0; // se va crea un nou arbore

  E key; float p;  // informatia din varful curent
  while ( f.read( (char *) &key, sizeof( key ) ) &&
          f.read( (char *) &p,   sizeof( p   ) )    )
    ins( key, p );

  f.close( );
  return 1;
}
```

constă în deschiderea fișierului binar cu numele dat de parametrul `char *file` prin intermediul unui obiect de tip `ifstream`, *input file stream*, citirea celor două componente ale fiecărui vârf (cheia `key` și frecvența `p`) și inserarea vârfului corespunzător în arbore. Neavând certitudinea că inițial arborele este vid, funcția de restaurare șterge toate vârfurile arborelui înainte de a începe inserarea cheilor citite din fișier.

Testarea corectitudinii operațiilor din clasele `ifstream` și `ofstream` se realizează prin invocarea implicită a operatorului de conversie la `int`. Acest operator returnează *false*, dacă starea stream-lui corespunde unei erori, sau *true*, în caz contrar. Invocarea lui este implicită, deoarece funcțiile membre `ifstream::read` și `ofstream::write` returnează obiectul invocator, iar sintaxa instrucțiunii `while` solicită o expresie de tip întreg. Acest operator de conversie la `int` este moștenit de la clasa `ios`, *input-output stream*, clasă din care sunt derivate toate celelalte clase utilizate pentru operațiile de intrare/ieșire.

9.6 Backtracking

Backtracking (în traducere aproximativă, “căutare cu revenire”) este un principiu fundamental de elaborare a algoritmilor pentru probleme de optimizare, sau de găsire a unor soluții care îndeplinesc anumite condiții. Algoritmii de tip backtracking se bazează pe o tehnică specială de explorare a grafurilor orientate implicite. Aceste grafuri sunt de obicei arbori, sau, cel puțin, nu conțin cicluri.

Pentru exemplificare, vom considera o problemă clasică: cea a plasării a opt regine pe tabla de șah, astfel încât nici una să nu intre în zona controlată de o alta. O metodă simplistă de rezolvare este de a încerca sistematic toate combinațiile

posibile de plasare a celor opt regine, verificând de fiecare dată dacă nu s-a obținut o soluție. Deoarece în total există

$$\binom{64}{8} = 4.426.165.368$$

combinații posibile, este evident că acest mod de abordare nu este practic. O primă îmbunătățire ar fi să nu plasăm niciodată mai mult de o regină pe o linie. Această restricție reduce reprezentarea pe calculator a unei configurații pe tabla de șah la un simplu vector, *posibil*[1 .. 8]: regina de pe linia i, $1 \le i \le 8$, se află pe coloana *posibil*[i], adică în poziția (i, *posibil*[i]). De exemplu, vectorul (3, 1, 6, 2, 8, 6, 4, 7) nu reprezintă o soluție, deoarece reginele de pe liniile trei și șase sunt pe aceeași coloană și, de asemenea, există două perechi de regine situate pe aceeași diagonală. Folosind această reprezentare, putem scrie în mod direct algoritmul care găsește o soluție a problemei:

```
procedure regine1
    for i1 ← 1 to 8 do
        for i2 ← 1 to 8 do
            ⋮
            for i8 ← 1 to 8 do
                posibil ← (i1, i2, ..., i8)
                if soluție(posibil) then write posibil
                                         stop
    write "nu există soluție"
```

De această dată, numărul combinațiilor este redus la 8^8 = 16.777.216, algoritmul oprindu-se de fapt după ce inspectează 1.299.852 combinații și găsește prima soluție.

Vom proceda acum la o nouă îmbunătățire. Dacă introducem și restricția ca două regine să nu se afle pe aceeași coloană, o configurație pe tabla de șah se poate reprezenta ca o permutare a primilor opt întregi. Algoritmul devine

```
procedure regine2
    posibil ← permutarea inițială
    while posibil ≠ permutarea finală and not soluție(posibil) do
        posibil ← următoarea permutare
    if soluție(posibil) then write posibil
                        else write "nu există soluție"
```

Sunt mai multe posibilități de a genera sistematic toate permutările primilor n întregi. De exemplu, putem pune fiecare din cele n elemente, pe rând, în prima poziție, generând de fiecare dată recursiv toate permutările celor n–1 elemente rămase:

```
procedure perm(i)
    if i = n   then  utilizează(T)   {T este o nouă permutare}
               else  for  j ← i  to n do  interschimbă T[i] și T[ j]
                                          perm(i+1)
                                          interschimbă T[i] și T[ j]
```

În algoritmul de generare a permutărilor, $T[1 .. n]$ este un tablou global inițializat cu $[1, 2, \ldots, n]$, iar primul apel al procedurii este *perm*(1). Dacă *utilizează*(*T*) necesită un timp constant, atunci *perm*(1) necesită un timp în $\Theta(n!)$.

Această abordare reduce numărul de configurații posibile la 8! = 40.320. Dacă se folosește algoritmul *perm*, atunci până la prima soluție sunt generate 2830 permutări. Mecanismul de generare a permutărilor este mai complicat decât cel de generare a vectorilor de opt întregi între 1 și 8. În schimb, verificarea faptului dacă o configurație este soluție se face mai ușor: trebuie doar verificat dacă nu există două regine pe aceeași diagonală.

Chiar și cu aceste îmbunătățiri, nu am reușit încă să eliminăm o deficiență comună a algoritmilor de mai sus: verificarea unei configurații prin "**if** *soluție*(*posibil*)" se face doar după ce toate reginele au fost deja plasate pe tablă. Este clar că se pierde astfel foarte mult timp.

Vom reuși să eliminăm această deficiență aplicând principiul backtracking. Pentru început, reformulăm problema celor opt regine ca o problemă de căutare într-un arbore. Spunem că vectorul $P[1 .. k]$ de întregi între 1 și 8 este *k-promițător*, pentru $0 \le k \le 8$, dacă zonele controlate de cele k regine plasate în pozițiile $(1, P[1]), (2, P[2]), \ldots, (k, P[k])$ sunt disjuncte. Matematic, un vector P este k-promițător dacă:

$$P[i] - P[j] \notin \{i - j, 0, j - i\}, \quad \text{pentru orice } 0 \le i,\ j \le k,\ i \ne j$$

Pentru $k \le 1$, orice vector P este k-promițător. Soluțiile problemei celor opt regine corespund vectorilor 8-promițători.

Fie V mulțimea vectorilor k-promițători, $0 \le k \le 8$. Definim graful orientat $G = \langle V, M \rangle$ astfel: $(P, Q) \in M$, dacă și numai dacă există un întreg k, $0 \le k \le 8$, astfel încât P este k-promițător, Q este $(k+1)$-promițător și $P[i] = Q[i]$ pentru fiecare $0 \le i \le k$. Acest graf este un arbore cu rădăcina în vectorul vid ($k = 0$). Vârfurile terminale sunt fie soluții ($k = 8$), fie vârfuri "moarte" ($k < 8$), în care este imposibil de plasat o regină pe următoarea linie fără ca ea să nu intre în zona controlată de reginele deja plasate. Soluțiile problemei celor opt regine se pot obține prin explorarea acestui arbore. Pentru aceasta, nu este necesar să generăm în mod explicit arborele: vârfurile vor fi generate și abandonate pe parcursul explorării. Vom parcurge arborele G în adâncime, ceea ce este echivalent aici cu o parcurgere în preordine, "coborând" în arbore numai dacă există șanse de a ajunge la o soluție.

Acest mod de abordare are două avantaje față de algoritmul *regine*2. În primul rând, numărul de vârfuri în arbore este mai mic decât 8!. Deoarece este dificil să calculăm teoretic acest număr, putem număra efectiv vârfurile cu ajutorul calculatorului: $\#V = 2057$. De fapt, este suficient să explorăm 114 vârfuri pentru a ajunge la prima soluție. În al doilea rând, pentru a decide dacă un vector este (k+1)-promițător, cunoscând că este extensia unui vector k-promițător, trebuie doar să verificăm ca ultima regină adăugată să nu fie pusă într-o poziție controlată de reginele deja plasate. Ca să apreciem cât am câștigat prin acest mod de verificare, să observăm că în algoritmul *regine*2, pentru a decide dacă o anumită permutare este o soluție, trebuia să verificăm fiecare din cele 28 de perechi de regine de pe tablă.

Am ajuns, în fine, la un algoritm performant, care afișează toate soluțiile problemei celor opt regine. Din programul principal, apelăm *regine*(0), presupunând că *posibil*[1 .. 8] este un tablou global.

```
procedure regine(k)
    {posibil[1 .. k] este k-promițător}
    if k = 8  then  write posibil {este o soluție}
              else  {explorează extensiile (k+1)-promițătoare
                     ale lui posibil}
                    for j ← 1 to 8 do
                        if plasare(k, j)  then  posibil[k+1] ← j
                                                regine(k+1)

function plasare(k, j)
    {returnează true, dacă și numai dacă se
     poate plasa o regină în poziția (k+1, j)}
    for i ← 1 to k do
        if  j–posibil[i] ∈ {k+1–i, 0, i–k–1} then return false
    return true
```

Problema se poate generaliza, astfel încât să plasăm n regine pe o tablă de n linii și n coloane. Cu ajutorul unor contraexemple, puteți arăta că problema celor n regine nu are în mod necesar o soluție. Mai exact, pentru $n \leq 3$ nu există soluție, iar pentru $n \geq 4$ există cel puțin o soluție.

Pentru valori mai mari ale lui n, avantajul metodei backtracking este, după cum ne și așteptăm, mai evident. Astfel, în problema celor douăsprezece regine, algoritmul *regine*2 consideră 479.001.600 permutări posibile și găsește prima soluție la a 4.546.044 configurație examinată. Arborele explorat prin algoritmul *regine* conține doar 856.189 vârfuri, prima soluție obținându-se deja la vizitarea celui de-al 262-lea vârf.

Algoritmii backtracking pot fi folosiți și atunci când soluțiile nu au în mod necesar aceeași lungime. Presupunând că nici o soluție nu poate fi prefixul unei alte soluții, iată schema generală a unui algoritm backtracking:

```
procedure backtrack(v[1 .. k])
    {v este un vector k-promițător}
    if v este o soluție
        then write v
        else for fiecare vector w care este (k+1)-promițător,
                    astfel încât w[1 .. k] = v[1 .. k]
             do backtrack(w[1 .. k+1])
```

Există foarte multe aplicații ale algoritmilor backtracking. Puteți încerca astfel rezolvarea unor probleme întîlnite în capitolele anterioare: problema colorării unui graf, problema 0/1 a rucsacului, problema monezilor (cazul general). Tot prin backtracking puteți rezolva și o variantă a problemei comis-voiajorului, în care admitem că există orașe fără legătură directă între ele și nu se cere ca ciclul să fie optim.

Parcurgerea în adâncime, folosită în algoritmul *regine*, devine și mai avantajoasă atunci când ne mulțumim cu o singură soluție a problemei. Sunt însă și probleme pentru care acest mod de explorare nu este avantajos.

Anumite probleme pot fi formulate sub forma explorării unui graf implicit care este infinit. În aceste cazuri, putem ajunge în situația de a explora fără sfârșit o anumită ramură infinită. De exemplu, în cazul cubului lui Rubik, explorarea manipulărilor necesare pentru a trece dintr-o configurație într-alta poate cicla la infinit. Pentru a evita asemenea situații, putem utiliza explorarea în lățime a grafului. În cazul cubului lui Rubik, mai avem astfel un avantaj: obținem în primul rând soluțiile care necesită cel mai mic număr de manipulări. Această idee este ilustrată de Exercițiul 9.15.

Am văzut că algoritmii backtracking pot folosi atât explorarea în adâncime cât și în lățime. Ceea ce este specific tehnicii de explorare backtracking este testul de fezabilitate, conform căruia, explorarea anumitor vârfuri poate fi abandonată.

9.7 Grafuri și jocuri

Cele mai multe jocuri strategice pot fi reprezentate sub forma grafurilor orientate în care vârfurile sunt poziții în joc, iar muchiile sunt mutări legale între două poziții. Dacă numărul pozițiilor nu este limitat a priori, atunci graful este infinit. Vom considera în cele ce urmează doar jocuri cu doi parteneri, fiecare având pe rând dreptul la o mutare. Presupunem, de asemenea, că jocurile sunt simetrice

(regulile sunt aceleași pentru cei doi parteneri) și deterministe (nu există un factor aleator).

Pentru a determina o strategie de câștig într-un astfel de joc, vom atașa fiecărui vârf al grafului o etichetă care poate fi de câștig, pierdere, sau remiză. Eticheta corespunde situației unui jucător care se află în poziția respectivă și trebuie să mute. Presupunem că nici unul din jucători nu greșește, fiecare alegând mereu mutarea care este pentru el optimă. În particular, din anumite poziții ale jocului nu se poate efectua nici o mutare, astfel de poziții terminale neavând poziții succesoare în graf. Etichetele vor fi atașate în mod sistematic astfel:

- Etichetele atașate unei poziții terminale depind de jocul în cauză. De obicei, jucătorul care se află într-o poziție terminală a pierdut.
- O poziție neterminală este o poziție de câștig, dacă cel puțin una din pozițiile ei succesoare în graf este o poziție de pierdere.
- O poziție neterminală este o poziție de pierdere, dacă toate pozițiile ei succesoare în graf sunt poziții de câștig.
- Orice poziție care a rămas neetichetată este o poziție de remiză.

Dacă jocul este reprezentat printr-un graf finit aciclic, această metodă etichetează vârfurile în ordine topologică inversă.

9.7.1 Jocul nim

Vom ilustra aceste idei printr-o variantă a jocului nim. Inițial, pe masă se află cel puțin două bețe de chibrit. Primul jucător ridică cel puțin un băț, lăsând pe masă cel puțin un băț. În continuare, pe rând, fiecare jucător ridică cel puțin un băț și cel mult de două ori numărul de bețe ridicate de către partenerul de joc la mutarea anterioară. Câștigă jucătorul care ridică ultimul băț. Nu există remize.

O poziție în acest joc este specificată atât de numărul de bețe de pe tablă, cât și de numărul maxim de bețe care pot fi ridicate la următoarea mutare. Vârfurile grafului asociat jocului sunt perechi $<i, j>$, $1 \le j \le i$, indicând că pot fi ridicate cel mult j bețe din cele i bețe de pe masă. Din vârful $<i, j>$ pleacă j muchii către vârfurile $<i–k, \min(2k, i–k)>$, $1 \le k \le j$. Vârful corespunzător poziției inițiale într-un joc cu n bețe, $n \ge 2$, este $<n, n–1>$. Toate vârfurile pentru care a două componentă este zero corespund unor poziții terminale, dar numai vârful $<0, 0>$ este interesant: vârfurile $<i, 0>$, pentru $i > 0$, sunt inaccesibile. În mod similar, vârfurile $<i, j>$, cu j impar și $j < i–1$, sunt inaccesibile. Vârful $<0, 0>$ corespunde unei poziții de pierdere.

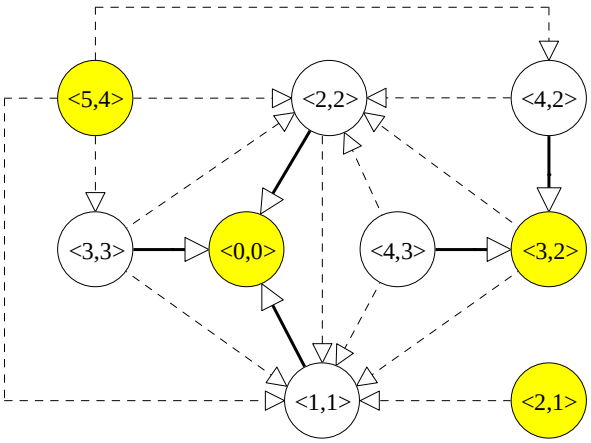


Figura 9.3 Graful unui joc.

Figura 9.3 reprezintă graful corespunzător jocului cu cinci bețe inițiale: vârfurile albe corespund pozițiilor de câștig, vârfurile gri corespund pozițiilor de pierdere, muchiile “continue” corespund mutărilor prin care se câștigă, iar muchiile “punctate” corespund mutărilor prin care se pierde. Dintr-o poziție de pierdere nu pleacă nici o muchie “continuă”, aceasta corespunzând faptului că din astfel de poziții nu există nici o mutare prin care se poate câștiga.

Se observă că jucătorul care are prima mutare într-un joc cu două, trei, sau cinci bețe nu are nici o strategie de câștig, dar are o astfel de strategie într-un joc cu patru bețe.

Următorul algoritm recursiv determină dacă o poziție este de câștig.

```
function rec(i, j)
    {returnează true dacă și numai dacă vârful
     <i, j> reprezintă o poziție de câștig;
     presupunem că 0 ≤ j ≤ i}
    for k ← 1 to j do
        if not rec(i–k, min(2k, i–k)) then return true
    return false
```

Algoritmul are același defect ca și algoritmul *fib*1 (Capitolul 1): calculează în mod repetat anumite valori. De exemplu, *rec*(5, 4) returnează *false* după ce a apelat succesiv

$$rec(4, 2),\ rec(3, 3),\ rec(2, 2),\ rec(1, 1)$$

Dar *rec*(3, 3) apelează, de asemenea, *rec*(2, 2) și *rec*(1, 1).

Putem evita acest lucru, construind prin programarea dinamică o matrice booleană globală, astfel încât $G[i, j] = true$, dacă și numai dacă $<i, j>$ este o poziție de câștig. Fie n numărul maxim de bețe folosite. Ca de obicei în programarea dinamică, calculăm matricea G de jos în sus:

```
procedure din(n)
    {calculează de jos în sus matricea G[1..n, 1..n]}
    G[0, 0] ← false
    for i ← 1 to n do
        for j ← 1 to i do
            k ← 1
            while k < j and G[i–k, min(2k, i–k)] do k ← k+1
            G[i, j] ← not G[i–k, min(2k, i–k)]
```

Prin tehnica programării dinamice, fiecare valoare a lui G este calculată o singură dată. Pe de altă parte însă, în acest context multe din valorile lui G sunt calculate în mod inutil. Astfel, este inutil să-l calculăm pe $G[i, j]$ atunci când j este impar și $j < i–1$. Iată și un alt exemplu de calcul inutil: știm că $<15, 14>$ este o poziție de câștig, imediat ce am aflat că al doilea succesor al său, $<13, 4>$, este o poziție de pierdere; valoarea lui $G(12, 6)$ nu mai este utilă în acest caz. Nu există însă nici un raționament "de jos în sus" pentru a nu-l calcula pe $G[12, 6]$. Pentru a-l calcula pe $G[15, 14]$, algoritmul *din* calculează 121 de valori $G[i, j]$, însă utilizează efectiv doar 27 de valori.

Algoritmul recursiv *rec* este ineficient, deoarece calculează anumite valori în mod repetat. Pe de altă parte, datorită raționamentului "de sus în jos", nu calculează niciodată valori pe care să nu le și utilizeze.

Rezultă că avem nevoie de o metodă care să îmbine avantajele formulării recursive cu cele ale programării dinamice. Cel mai simplu este să adăugăm

algoritmului recursiv o *funcție de memorie* care să memoreze dacă un vârf a fost deja vizitat sau nu. Pentru aceasta, definim matricea booleană globală *init*[0 .. *n*, 0 .. *n*], inițializată cu *false*.

```
function nim(i, j)
    if init[i, j] then return G[i, j]
    init[i, j] ← true
    for k ← 1 to j do
        if not nim(i–k, min(2k, i–k)) then  G[i, j] ← true
                                            return true
    G[i, j] ← false
    return false
```

Deoarece matricea *init* trebuie inițializată, aparent nu am câștigat nimic față de algoritmul care folosește programarea dinamică. Avantajul obținut este însă mare, deoarece operația de inițializare se poate face foarte eficient, după cum vom vedea în Secțiunea 10.2.

Când trebuie să soluționăm mai multe cazuri similare ale aceleiași probleme, merită uneori să calculăm câteva rezultate auxiliare care să poată fi apoi folosite pentru accelerarea soluționării fiecărui caz. Această tehnică se numește *precondiționare* și este exemplificată în Exercițiul 9.7.

Jocul nim este suficient de simplu pentru a permite și o rezolvare mai eficientă decât prin algoritmul *nim*, fără a folosi graful asociat. Algoritmul de mai jos determină strategia de câștig folosind precondiționarea. Într-o poziție inițială cu n bețe, se apelează la început *precond*(n). Se poate arăta că un apel *precond*(n) necesită un timp în $\Theta(n)$. După aceea, orice apel *mutare*(i, j), $1 \leq j \leq i$, returnează într-un timp în $\Theta(1)$ câte bețe să fie ridicate din poziția $<i, j>$, pentru o mutare de câștig. Dacă poziția $<i, j>$ este de pierdere, în mod convențional se indică ridicarea unui băț, ceea ce întîrzie pe cât posibil pierderea inevitabilă a jocului. Tabloul T [0 .. n] este global.

```
procedure precond(n)
    T[0] ← ∞
    for i ← 1 to n do
        k ← 1
        while T[i–k] ≤ 2k do k ← k+1
        T[i] ← k

function mutare(i, j)
    if j < T[i] then return 1 {prelungește agonia!}
    return T[i]
```

Nu vom demonstra aici corectitudinea acestui algoritm.

9.7.2 Șahul și tehnica minimax

Șahul este, desigur, un joc mult mai complex decât jocul nim. La prima vedere, graful asociat șahului conține cicluri. Există însă reglementări ale Federației Internaționale de Șah care previn intrarea într-un ciclu. De exemplu, se declară remiză o partidă după 50 de mutări în care nu are loc nici o acțiune ireversibilă (mutarea unui pion, sau eliminarea unei piese). Datorită acestor reguli, putem considera că graful asociat șahului nu are cicluri.

Vom eticheta fiecare vârf ca poziție de câștig pentru Alb, poziție de câștig pentru Negru, sau remiză. Odată construit, acest graf ne permite să jucăm perfect șah, adică să câștigăm mereu, când este posibil, și să pierdem doar când este inevitabil. Din nefericire (din fericire pentru jucătorii de șah), acest graf conține atâtea vârfuri, încât nu poate fi explorat complet nici cu cel mai puternic calculator existent.

Deoarece o căutare completă în graful asociat jocului de șah este imposibilă, nu putem folosi tehnica programării dinamice. Se impune atunci, în mod natural, aplicarea unei tehnici recursive, care să modeleze raționamentul “de sus în jos”. Această tehnică (numită *minimax*) este de tip euristic, și nu ne oferă certitudinea câștigării unei partide. Ideea de bază este următoarea: fiind într-o poziție oarecare, se alege una din cele mai bune mutări posibile, explorând doar o parte a grafului. Este de fapt o modelare a raționamentului unui jucător uman care gândește doar cu un mic număr de mutări în avans.

Primul pas este să definim o funcție de evaluare statică *eval*, care atribuie o anumită valoare fiecărei poziții posibile. În mod ideal, $eval(u)$ va crește atunci când poziția u devine mai favorabilă Albului. Această funcție trebuie să țină cont de mai mulți factori: numărul și calitatea pieselor existente de ambele părți, controlul centrului tablei, libertatea de mișcare etc. Trebuie să facem un compromis între acuratețea acestei funcții și timpul necesar calculării ei. Când se aplică unei poziții terminale, funcția de evaluare trebuie să returneze $+\infty$ dacă a câștigat Albul, $-\infty$ dacă a câștigat Negrul și 0 dacă a fost remiză.

Dacă funcția de evaluare statică ar fi perfectă, ar fi foarte ușor să determinăm care este cea mai bună mutare dintr-o anumită poziție. Să presupunem că este rândul Albului să mute din poziția u. Cea mai bună mutare este cea care îl duce în poziția v, pentru care

$$eval(v) = \max\{eval(w) \mid w \text{ este succesor al lui } u\}$$

Această poziție se determină astfel:

$val \leftarrow -\infty$
for fiecare w succesor al lui u **do**
  if $eval(w) \geq val$ **then** $val \leftarrow eval(w)$
    $v \leftarrow w$

Complexitatea jocului de șah este însă atât de mare încât este imposibil să găsim o astfel de funcție de evaluare perfectă.

Presupunând că funcția de evaluare nu este perfectă, o strategie bună pentru Alb este să prevadă că Negrul va replica cu o mutare care minimizează funcția *eval*. Albul gândește astfel cu o mutare în avans, iar funcția de evaluare este calculată în mod dinamic.

```
val ← −∞
for fiecare w succesor al lui u do
    if w nu are succesor
        then  valw ← eval(w)
        else  valw ← min{eval(x) | x este succesor al lui w}
    if valw ≥ val  then  val ← valw
                         v ← w
```

Pentru a adăuga și mai mult dinamism funcției *eval*, este preferabil să investigăm mai multe mutări în avans. Din poziția *u*, analizând *n* mutări în avans, Albul va muta atunci în poziția *v* dată de

```
val ← −∞
for fiecare w succesor al lui u do
    if negru(w, n) ≥ val      then  val ← negru(w, n)
                                    v ← w
```

Funcțiile *negru* și *alb* sunt următoarele:

```
function negru(w, n)
    if n = 0 or w nu are succesori
        then return eval(w)
    return min{alb(x, n−1) | x este succesor al lui w}

function alb(x, n)
    if n = 0 or x nu are succesori
        then return eval(x)
    return max{negru(w, n−1) | w este succesor al lui x}
```

Acum înțelegem de ce această tehnică este numită minimax: Negrul încearcă să minimizeze avantajul pe care îl permite Albului, iar Albul încearcă să maximizeze avantajul pe care îl poate obține la fiecare mutare.

Tehnica minimax poate fi îmbunătățită în mai multe feluri. Astfel, explorarea anumitor ramuri poate fi abandonată mai curând, dacă din informația pe care o deținem asupra lor, deducem că ele nu mai pot influența valoarea vârfurilor situate la un nivel superior. Această îmbunătățire se numește *retezare alfa-beta* (*alpha-beta pruning*) și este exemplificată în Figura 9.4. Presupunând că valorile numerice atașate vârfurilor terminale sunt valorile funcției *eval* calculate în

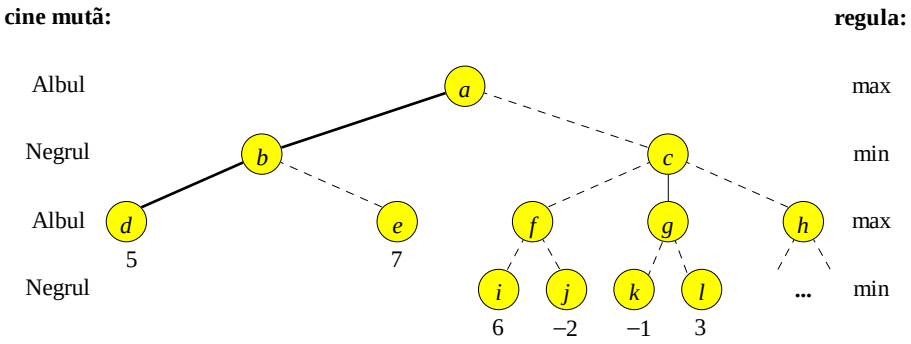


Figura 9.4 Retezare alfa-beta.

poziţiile respective, celelalte valori se pot calcula prin tehnica minimax, parcurgând arborele în postordine. Obţinem succesiv *eval*(*b*) = 5, *eval*(*f*) = 6, *eval*(*g*) = 3. În acest moment ştim deja că *eval*(*c*) ≤ 3 şi, fără să-l mai calculăm pe *eval*(*h*), obţinem valoarea *eval*(*a*) = 5. Cu alte cuvinte, la o anumită fază a explorării am dedus că putem abandona explorarea subarborelui cu rădăcina în *h* (îl putem "reteza").

Tehnica minimax determină în final strategia reprezentată în Figura 9.4 prin muchiile continue.

9.8 Grafuri AND/OR

Multe probleme se pot descompune într-o serie de subprobleme, astfel încât rezolvarea tuturor acestor subprobleme, sau a unora din ele, să ducă la rezolvarea problemei iniţiale. Descompunerea unei probleme complexe, în mod recursiv, în subprobleme mai simple poate fi reprezentată printr-un graf orientat. Această descompunere se numeşte *reducerea problemei* şi este folosită în demonstrarea automată, integrare simbolică şi, în general, în inteligenţa artificială. Într-un graf orientat de acest tip vom permite unui vârf neterminal *v* oarecare două alternative. Vârful *v* este de tip **AND** dacă reprezintă o problemă care este rezolvată doar dacă *toate* subproblemele reprezentate de vârfurile adiacente lui *v* sunt rezolvate. Vârful *v* este de tip **OR** dacă reprezintă o problemă care este rezolvată doar dacă *cel puţin* o subproblemă reprezentată de vârfurile adiacente lui *v* este rezolvată. Un astfel de graf este de tip **AND/OR**.

De exemplu, arborele **AND/OR** din Figura 9.5 reprezintă reducerea problemei *A*. Vârfurile terminale reprezintă probleme primitive, marcate ca rezolvabile (vârfurile albe), sau nerezolvabile (vârfurile gri). Vârfurile neterminale reprezintă

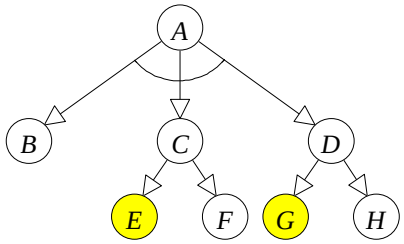


Figura 9.5 Un arbore **AND**/**OR**.

probleme despre care nu se știe a priori dacă sunt rezolvabile sau nerezolvabile. Vârful *A* este un vârf **AND** (marcăm aceasta prin unirea muchiilor care pleacă din *A*), vârfurile *C* și *D* sunt vârfuri **OR**. Să presupunem acum că dorim să aflăm dacă problema *A* este rezolvabilă. Deducem succesiv că problemele *C*, *D* și *A* sunt rezolvabile.

Într-un arbore oarecare **AND**/**OR**, următorul algoritm determină dacă problema reprezentată de un vârf oarecare *u* este rezolvabilă sau nu. Un apel *sol*(*u*) are ca efect parcurgerea în postordine a subarborelui cu rădăcina în *u* și returnarea valorii *true*, dacă și numai dacă problema este rezolvabilă.

```
function sol(v)
   case
      v este terminal:      if v este rezolvabil
                               then  return true
                               else  return false
      v este un vîrf AND:   for fiecare vîrf w adiacent lui v do
                               if not sol(w) then return false
                            return true
      v este un vîrf OR:    for fiecare vîrf w adiacent lui v do
                               if sol(w) then return true
                            return false
```

Ca și în cazul retezării alfa-beta, dacă în timpul explorării se poate deduce că un vârf este rezolvabil sau nerezolvabil, se abandonează explorarea descendenților săi. Printr-o modificare simplă, algoritmul *sol* poate afișa strategia de rezolvare a problemei reprezentate de *u*, adică subproblemele rezolvabile care conduc la rezolvarea problemei din *u*.

Cu anumite modificări, algoritmul se poate aplica asupra grafurilor **AND**/**OR** oarecare. Similar cu tehnica backtracking, explorarea se poate face atât în adâncime (ca în algoritmul *sol*), cât și în lățime.

9.9 Exerciții

9.1 Într-un arbore binar de căutare, care este modul de parcurgere a vârfurilor pentru a obține lista ordonată crescător a cheilor?

9.2 Fiecărei expresii aritmetice în care apar numai operatori binari i se poate atașa în mod natural un arbore binar. Dați exemple de parcurgere în inordine, preordine și postordine a unui astfel de arbore. Se obțin diferite moduri de scriere a expresiilor aritmetice. Astfel, parcurgerea în postordine generează scrierea postfixată menționată în Secțiunea 3.1.1.

9.3 Fie un arbore binar reprezentat prin adrese, astfel încât vârful i (adică vârful a cărui adresă este i) este memorat în trei locații diferite conținând:

$VAL[i]$ = valoarea vârfului
$ST[i]$ = adresa fiului stâng
$DR[i]$ = adresa fiului drept

(Dacă se folosește o implementare prin tablouri paralele, atunci adresele sunt indici de tablou). Presupunem că variabila *root* conține adresa rădăcinii arborelui și că o adresă este zero, dacă și numai dacă vârful către care se face trimiterea lipsește. Scrieți algoritmii de parcurgere în inordine, preordine și postordine a arborelui. La fiecare consultare afișați valoarea vârfului respectiv.

Soluție: Pentru parcurgerea în inordine apelăm *inordine*(*root*), *inordine* fiind procedura

```
procedure inordine(i)
    if i ≠ 0 then
        inordine(ST[i])
        write VAL[i]
        inordine(DR[i])
```

9.4 Dați un algoritm care folosește parcurgerea *i)* în adâncime *ii)* în lățime pentru a afla numărul componentelor conexe ale unui graf neorientat. În particular, puteți determina astfel dacă graful este conex. Faceți o comparație cu algoritmul din Exercițiul 3.12.

9.5 Într-un graf orientat, folosind principiul parcurgerii în lățime, elaborați un algoritm care găsește cel mai scurt ciclu care conține un anumit vârf dat. În locul parcurgerii în lățime, puteți folosi parcurgerea în adâncime?

9.6 Revedeți Exercițiul 8.8. Scrieți un algoritm care găsește închiderea tranzitivă a unui graf orientat. Folosiți parcurgerea în adâncime sau lățime. Comparați algoritmul obținut cu algoritmul lui Warshall.

9.7 Într-un arbore cu rădăcină, elaborați un algoritm care verifică pentru două vârfuri oarecare *v* și *w*, dacă *w* este un descendent al lui *v*. (Pentru ca problema să nu devină trivială, presupunem că vârfurile nu conțin adresa tatălui).

Indicație: Orice soluție directă necesită un timp în $\Omega(n)$, în cazul cel mai nefavorabil, unde *n* este numărul vârfurilor subarborelui cu rădăcina în *v*.

Iată un mod indirect de rezolvare a problemei, care este în principiu avantajos atunci când trebuie să verificăm mai multe cazuri (perechi de vârfuri) pentru același arbore. Fie *preord*[1 .. *n*] și *postord*[1 .. *n*] tablourile care conțin ordinea de parcurgere a vârfurilor în preordine, respectiv în postordine. Pentru oricare două vârfuri *v* și *w* avem:

preord[*v*] < *preord*[*w*] ⇔ *w* este un descendent al lui *v*, sau *v* este la stânga lui *w* în arbore

postord[*v*] > *postord*[*w*] ⇔ *w* este un descendent al lui *v*, sau *v* este la dreapta lui *w* în arbore

Deci, *w* este un descendent al lui *v*, dacă și numai dacă:

$$preord[v] < preord[w] \quad \text{și} \quad postord[v] > postord[w]$$

După ce calculăm valorile *preord* și *postord* într-un timp în $\Theta(n)$, orice caz particular se poate rezolva într-un timp în $\Theta(1)$. Acest mod indirect de rezolvare ilustrează metoda precondiționării.

9.8 Fie *A* arborele parțial generat de parcurgerea în adâncime a grafului neorientat conex *G*. Demonstrați că, pentru orice muchie {*v*, *w*} din *G*, este adevărată următoarea proprietate: *v* este un descendent sau un ascendent al lui *w* în *A*.

Soluție: Dacă muchiei {*v*, *w*} îi corespunde o muchie în *A*, atunci proprietatea este evident adevărată. Putem presupune deci că vârfurile *v* și *w* nu sunt adiacente în *A*. Fără a pierde din generalitate, putem considera că *v* este vizitat înaintea lui *w*. Parcurgerea în adâncime a grafului *G* înseamnă, prin definiție, că explorarea vârfului *v* nu se încheie decât după ce a fost vizitat și vârful *w* (ținând cont de existența muchiei {*v*, *w*}). Deci, *v* este un ascendent al lui *w* în *A*. **9.9** Dacă *v* este un vârf al unui graf conex, demonstrați că *v* este un punct de articulare, dacă și numai dacă există două vârfuri *a* și *b* diferite de *v*, astfel încât orice drum care îl conectează pe *a* cu *b* trece în mod necesar prin *v*.

9.10 Fie G un graf neorientat conex, dar nu și biconex. Elaborați un algoritm pentru găsirea mulțimii minime de muchii care să fie adăugată lui G, astfel încât G să devină biconex. Analizați algoritmul obținut.

9.11 Fie $M[1 .. n, 1 .. n]$ o matrice booleană care reprezintă un labirint în forma unei table de șah. În general, pornind dintr-un punct dat, este permis să mergeți către punctele adiacente de pe aceeași linie sau coloană. Prin punctul (i, j) se poate trece dacă și numai dacă $M(i, j)$ este *true*. Elaborați un algoritm backtracking care găsește un drum între colțurile (1, 1) și (n, n), dacă un astfel de drum există.

9.12 În algoritmul *perm* de generare a permutărilor, înlocuiți "*utilizează*(T)" cu "**write** T " și scrieți rezultatul afișat de *perm*(1), pentru $n = 3$. Faceți apoi același lucru, presupunând că tabloul T este inițializat cu $[n, n-1, \ldots, 1]$.

9.13 (*Problema submulțimilor de sumă dată*). Fie mulțimea de numere pozitive $W = \{w_1, \ldots, w_n\}$ și fie M un număr pozitiv. Elaborați un algoritm backtracking care găsește toate submulțimile lui W pentru care suma elementelor este M.

Indicație: Fie $W = \{11, 13, 24, 7\}$ și $M = 31$. Cel mai important lucru este cum reprezentăm vectorii care vor fi vârfurile arborelui generat. Iată două moduri de reprezentare pentru soluția (11, 13, 7):

i) Prin vectorul indicilor: (1, 2, 4). În această reprezentare, vectorii soluție au lungimea variabilă.

ii) Prin vectorul boolean $x = (1, 1, 0, 1)$, unde $x[i] = 1$, dacă și numai dacă w_i a fost selectat în soluție. De această dată, vectorii soluție au lungimea constantă.

9.14 Un cal este plasat în poziția arbitrară (i, j), pe o tablă de șah de $n \times n$ pătrate. Concepeți un algoritm backtracking care determină n^2-1 mutări ale calului, astfel încât fiecare poziție de pe tablă este vizitată exact o dată (presupunând că o astfel de secvență de mutări există).

9.15 Găsiți un algoritm backtracking capabil să transforme un întreg n într-un întreg m, aplicând cât mai puține transformări de forma $f(i) = 3i$ și $g(i) = \lfloor i/2 \rfloor$. De exemplu, 15 poate fi transformat în 4 folosind patru transformări: $4 = gfgg(15)$. Cum se comportă algoritmul dacă este imposibil de transformat astfel n în m ?

9.16 Modificați algoritmul *rec* pentru jocul nim, astfel încât să returneze un întreg k:

i) $k = 0$, dacă poziția este de pierdere.

ii) $1 \leq k \leq j$, dacă "a lua k bețe" este o mutare de câștig.

9.17 Jocul lui Grundy seamănă foarte mult cu jocul nim. Inițial, pe masă se află o singură grămadă de n bețe. Cei doi jucători au alternativ dreptul la o mutare. O mutare constă din împărțirea uneia din grămezile existente în două grămezi de mărimi diferite (dacă acest lucru nu este posibil, adică dacă toate grămezile constau din unul sau două bețe, jucătorul pierde partida). Ca și la nim, remiza este exclusă. Găsiți un algoritm care să determine dacă o poziție este de câștig sau de pierdere.

9.18 Tehnica minimax modelează eficient, dar în același timp și simplist, comportamentul unui jucător uman. Una din presupunerile noastre a fost că nici unul dintre jucători nu greșește. În ce măsură rămîne valabilă această tehnică dacă admitem că: *i)* jucătorii pot să greșească, *ii)* fiecare jucător nu exclude posibilitatea ca partenerul să facă greșeli.

9.19 Dacă graful obținut prin reducerea unei probleme are și vârfuri care nu sunt de tip **AND** sau de tip **OR**, arătați că prin adăugarea unor vârfuri fictive putem transforma acest graf într-un graf **AND/OR**.

9.20 Modificați algoritmul *sol* pentru a-l putea aplica grafurilor **AND/OR** oarecare.

10. Derivare publică, funcții virtuale

Derivarea publică și funcțiile virtuale sunt mecanismele esențiale pentru programarea orientată pe obiect în limbajul C++. Cele două exemple prezentate în acest capitol au fost alese pentru a ilustra nu numai eleganța deosebită a programării orientate pe obiect, ci și problemele care pot apare atunci când se fac greșeli de proiectare.

10.1 Ciurul lui Eratostene

Acest exemplu este construit pornind de la un cunoscut algoritm pentru determinarea numerelor prime. Geograful și astronomul grec Eratostene din Cirena (sec. III a.Ch.) a avut ideea de a transforma proprietatea numerelor prime de a nu fi multiplii nici unuia din numerele mai mici decât ele, într-un criteriu de selecție (cernere): din șirul primelor primelor *n* numere naturale se elimină pe rând multiplii lui 2, 3, 5 etc, elementele rămase fiind cele prime. Astfel, 2 fiind prim, din șir se elimină multiplii lui 2 adică 4, 6, 8 etc. Următorul număr rămas este 3, deci 3 este prim și se vor elimina numerele 9, 15, 21 etc, multiplii pari ai lui 3 fiind deja eliminați. Și așa mai departe, până la determinarea tuturor numerelor prime mai mici decât un număr dat.

Implementarea ciurului lui Eratostene prezentată în continuare[*] nu este foarte eficientă ca timp de execuție și memorie utilizată. În schimb, este atât de "orientată pe obiect", încât merită să fie prezentată ca una din cele mai tipice aplicații C++. Ciurul este construit dintr-un șir de *site* și un generator de numere numit *contor*. Fiecare sită corespunde unui număr prim. Ea solicită valori (numere) de cernut de la sita următoare și lasă să treacă, returnând sitei anterioare, doar acele valori care nu sunt multipli ai numărului prim corespunzător sitei. Ultimul element în această structură de site este contorul, care nu face decât să genereze numere, rând pe rând. Primul element este ciurul propriu-zis, din care vor ieși doar numere prime. În plus, ciurul mai are sarcina de a crea sita corespunzătoare numărului prim tocmai determinat.

La început, avem doar ciurul și contorul, acesta din urmă inițializat cu valoarea 2 (Figura 10.1). Prima valoare extrasă din ciur este și prima valoare returnată de

[*] Implementarea este preluată din R. Sethi, "*Programming Languages. Concepts and Constructs*", Secțiunea 6.7.

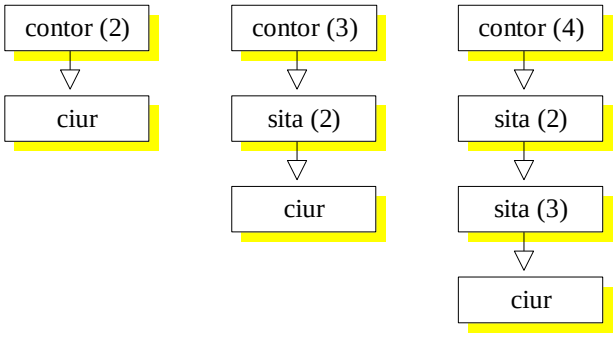


Figura 10.1 Ciurul lui Eratostene.

contor, și anume 2. După această primă iterație, contorul va avea valoarea 3, iar între ciur și contor se va insera prima sită, sită corespunzătoare lui 2. Ciurul va solicita o nouă valoare sitei 2 care, la rândul ei, va solicita o nouă valoare contorului. Contorul emite 3, schimbându-și valoarea la 4, 3 va trece prin sita 2 și va ajunge la ciur. Imediat, sita 3 se inserează în lista existentă.

Contorul, la solicitarea ciurului, solicitare transmisă sitei 3, apoi sitei 2, va returna în continuare 4. Valoarea 4 nu trece de sita 2, dar sita 2 insistă, căci trebuie să răspundă solicitării primite, astfel încât va primi un 5. Această valoare trece de toate sitele și lista are un nou element. Continuând procesul, constatăm că 6 se blochează la sita 2, 7 trece prin toate sitele (5, 3, 2), iar valorile 8 și 9 sunt blocate de sitele 2, respectiv 3. La un moment dat, contorul va avea valoarea *n*, iar în listă vor fi sitele corespunzătoare tuturor numerelor prime mai mici decât *n*.

Pentru implementarea acestui comportament, avem nevoie de o listă înlănțuită, în care fiecare element este sursă de valori pentru predecesor și își cunoaște propria sursă (elementul succesor). Altfel spus, fiecare element are cel puțin doi membri: adresa sursei și funcția care cerne valorile.

```
class element {
public:
  element( element *src ) { sursa = src; }
  virtual int cerne( ) { return 0; }

protected:
  element *sursa;
};
```

Acest `element` este un prototip, deoarece lista conține trei tipuri diferite de elemente, diferențiate prin funcția de cernere:

- Ciurul, care creează site.
- Sitele, care cern valorile.
- Contorul, care doar generează valori.

Cele trei tipuri fiind particularizări ale tipului `element`, le vom deriva public din acesta, creând astfel trei subtipuri.

```
class contor: public element {
public:
  contor( int v ): element( 0 ) { valoare = v; }
  int cerne( ) { return valoare++; };

private:
  int valoare;
};

class ciur: public element {
public:
  ciur( element *src ): element( src ) { }
  int cerne( );
};

class sita: public element {
public:
  sita( element *src, int f ): element(src) { factor = f; }
  int cerne( );

private:
  int factor;
};
```

Clasa `contor` este definită complet. Primul element generat (“cernut”) este stabilit prin `v`, parametrul constructorului. Pentru clasa `sita`, funcția de cernere este mult mai selectivă. Ea solicită de la sursă valori, până când primește o valoare care nu este multiplu al propriei valori.

```
int sita::cerne( ) {
  while ( 1 ) {
    int n = sursa->cerne( );
    if ( n % factor ) return n;
  }
}
```

Pentru `ciur`-ul propriu-zis, funcția de cernere nu mai are nimic de cernut. Valoarea primită de la `sita` sursă este în mod sigur un număr prim, motiv pentru care `ciur`-ul va crea `sita` corespunzătoare.

```
int ciur::cerne( ) {
 int n = sursa->cerne( );
 sursa = new sita( sursa, n );
 return n;
}
```

Se observă că noua sită este creată și inserată în ciur printr-o singură instrucțiune:

```
 sursa = new sita( sursa, n );
```

al cărei efect poate fi exprimat astfel: sursa nodului `ciur` este o nouă `sita` (cu valoarea `n`) a cărei sursă va fi sursa actuală a ciurului. O astfel de operație de inserare este una dintre cele mai uzuale în lucrul cu liste.

Prin programul următor, ciurul descris poate fi pus în funcțiune pentru determinarea numerelor prime mai mici decât o anumită valoare.

```
#include <iostream.h>
#include <stdlib.h>
#include <new.h>

// definitiile claselor element, contor, ciur, sita

void no_mem( ) { cerr << "no_mem"; exit( 1 ); }

int main( ) {
  _new_handler = no_mem;

  int max;
  cout << "max ... "; cin >> max;

  ciur s( &contor( 2 ) );
  int prim;

  do {
    prim = s.cerne( );
    cout << prim << ' ';
  } while ( prim < max );
  cout << '\n';

  return 0;
}
```

Înainte de a introduce valori `max` prea mari, este bine să vă asigurați că stiva programului este suficient de mare și că aveți suficient de multă memorie liberă pentru a fi alocată dinamic.

Folosind cunoștințele expuse până acum, această ciudată implementare a algoritmului lui Eratostene nu are cum să funcționeze. Iată cel puțin două motive:

- În instrucțiunea `int n = sursa->cerne( )` din clasele `sita` și `ciur`, membrul `sursa`, moștenit de la clasa de bază `element`, este de tip pointer la `element`, deci funcția `cerne()` invocată este cea definită în clasa `element`. În consecință, ceea ce se obține ar trebui să fie un șir infinit de `0`-uri (desigur, în limita memoriei disponibile).
- Există o nepotrivire între argumentele formale (de tip pointer la `element`) ale constructorilor claselor `ciur`, `sita` și argumentele actuale cu care sunt invocați acești constructori. Astfel, constructorul clasei `ciur` este invocat cu un pointer la `contor`, iar constructorul clasei `sita` este invocat prima dată cu un pointer la `contor` și apoi cu pointeri la `sita`.

Elementele esențiale în elucidarea acestor aspecte sunt derivările publice și definiția din clasa `element`:

```
virtual int cerne( ) { return 0; }
```

Prin derivarea `public`, tipurile `ciur`, `sita` și `contor` sunt subtipuri ale tipului de bază `element`. Conversia de la un subtip (tip derivat public) la tipul de bază este o conversie sigură, bine definită. Membrii tipului de bază vor fi totdeauna corect inițializați cu valorile membrilor respectivi din subtip, iar valorile membrilor din subtip, care nu se regăsesc în tipul de bază, se vor pierde. Din aceleași motive, conversia subtip-tip de bază se extinde și asupra pointerilor sau referințelor. Altfel spus, în orice situație, un obiect, un pointer sau o referință la un obiect dintr-un tip de bază, poate fi înlocuit cu un obiect, un pointer sau o referință la un obiect dintr-un tip derivat public.

Declarația `virtual` din funcția `cerne()` permite implementarea *legăturilor dinamice*. Prin redefinirea funcției `cerne()` în fiecare din subtipurile derivate din `element`, se permite invocarea diferențiată a funcțiilor `cerne()` printr-o sintaxă unică:

```
sursa->cerne( )
```

Dacă `sursa` este de tip pointer la `element`, atunci, după cum am precizat mai sus, oricând este posibil ca obiectul `sursa` să fie dintr-un tip derivat din `element`. Funcția `cerne()` fiind virtuală în tipul de bază, funcția efectiv invocată nu va fi cea din tipul de bază, ci cea din tipul actual al obiectului invocator. Acest mecanism implică stabilirea unei legături dinamice, în timpul execuției programului, între tipul actual al obiectului invocator și funcția virtuală.

Datorită faptului că definiția din clasa de bază a funcției `cerne()`, practic nu are importanță, este posibil să o lăsăm nedefinită:

```
virtual int cerne( ) = 0;
```

Consecința acestei acțiuni este că nu mai putem defini obiecte de tip `element`, clasa putând servi doar ca bază pentru derivarea unor subtipuri. Astfel de funcții se numesc *virtuale pure*, iar clasele respective sunt numite *clase abstracte*.

10.2 Tablouri inițializate virtual

Tabloul, una din cele mai simple structuri de date, are drept caracteristică principală adresarea elementelor sale în timp constant. Regăsirea sau modificarea valorii unui element de tablou sunt operații care necesită un timp constant, independent de factori cum ar fi poziția (indicele) elementului, sau faptul că este neinițializat, inițializat sau chiar modificat de mai multe ori.

Adresarea elementelor pentru citirea sau modificarea valorilor lor, operație numită *indexare*, este considerată operația fundamentală asociată structurii de tablou. Chiar dacă este ușor de implementat, cu un timp de execuție constant (vezi operatorul de indexare din clasa `tablou<T>`, Secțiunea 4.1.3), uneori indexarea se dovedește a fi costisitoare. Algoritmul *nim* ne-a pus în fața unei astfel de situații: tabloul *init* este util doar dacă poate fi nu numai adresat, ci și inițializat în timp constant, timp imposibil de atins în condițiile inițializării fiecărui element prin operatorul de indexare. Aparent, tabloul nu se pretează la o astfel de inițializare și deci ar fi bine să lucrăm cu o altă structură. De exemplu, cu o listă a elementelor modificate. Valoarea oricărui element este cea memorată în listă, sau este o valoare implicită, dacă el nu apare aici. Inițializarea nu mai depinde, în această situație, de numărul elementelor, dar nici adresarea nu mai este o operație cu timp constant de execuție!

Inițializarea unui tablou în timp constant, împreună cu accesarea elementelor tot în timp constant, sunt două cerințe aparent contradictorii pentru structura de tablou. Eliminarea contradicției, în caz că este posibilă (și este), impune completarea tabloului cu o nouă operație elementară, *inițializarea*, precum și modificarea corespunzătoare a operatorului de indexare. Obținem un alt tip de tablou, în care elementele nu mai sunt inițializate efectiv, fiecare în parte, ci virtual, printr-un operator de inițializare globală.

În continuare, vom prezenta o structură de *tablou inițializat virtual*[*], precum și implementarea corespunzătoare în limbajul C++.

---

[*] Ideea acestei structuri este sugerată în A. V. Aho, J. E. Hopcroft și J. D. Ullman, *"The Design and Analysis of Computer Algorithms"*.

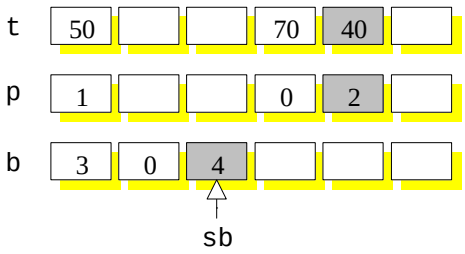


Figura 10.2 Structura de tablou inițializat virtual.

10.2.1 Structura

Întreaga construcție se bazează pe o idee similară cu mai sus menționata listă a elementelor modificate. Este o idee cât se poate de naturală, completată cu o ingenioasă modalitate de evitare a parcurgerii secvențiale a listei.

Tabloului inițializat virtual `a` i se asociază un tablou `b`, în care sunt memorate pozițiile elementelor modificate. De exemplu, dacă s-au modificat elementele `a[3]`, `a[0]` și `a[4]`, atunci primele trei poziții din `b` au valorile `3`, `0` și `4`. Se observa că `b` este o listă implementată secvențial, sub forma unui tablou.

Dacă nici o poziție din `b` nu a fost ocupată, atunci orice element din `a` are valoarea implicită (stabilită apriori). Dacă `b` este complet ocupat, atunci toate elementele din `a` au fost modificate, deci valorile lor pot fi obținute direct din locațiile corespunzătoare din `a`. Pentru evidența locațiilor ocupate din lista `b` (și implicit a celor libere), vom folosi variabila `sb`, variabilă inițializată cu `-1`.

Cum procedăm în cazul general, când doar o parte din elemente au fost modificate? Nu vom parcurge locațiile ocupate din `b`, ci va trebui să decidem în timp constant dacă un anumit element, de exemplu `a[1]`, a fost sau nu a fost modificat. Pentru a atinge această “performanță”, vom completa structura cu un alt tablou, tabloul `p`, în care sunt memorate pozițiile din `b` ale fiecărui element modificat (Figura 10.2).

Tabloul `p`, tablou paralel cu tabloul `a`, se inițializează pe măsură ce elementele din `a` sunt modificate. În exemplul nostru, `p[3]` este `0` deoarece elementul `a[3]` a fost modificat primul, ocupând astfel prima poziție în `b`. Apoi, `p[0]` este `1` pentru că `a[0]` este al doilea element modificat, iar `p[4]` are valoarea `2` deoarece al treilea element modificat a fost `a[4]`. Valoarea lui `a[1]`, valoare nemodificată, se obține prin intermediul tablourilor `p` și `b` astfel:

- Dacă `p[1]` nu este o poziție validă în `b`, adică `0` > `p[1]` sau `p[1]` > `sb`, atunci `a[1]` nu a fost modificat și are valoarea implicită.
- Dacă `p[1]` este o poziție validă în `b`, atunci, pentru ca `a[1]` să nu aibă valoarea implicită, `b[p[1]]` trebuie să fie `1`. Deoarece `a[1]` nu a fost modificat, nici o poziție ocupată din `b` nu are însă valoarea `1`. Deci, `a[1]` are valoarea implicită.

Să vedem acum ce se întâmplă pentru un element deja modificat, cum este `a[0]`. Valoarea lui `p[0]` corespunde unei poziții ocupate din `b`, iar `b[p[0]]` este `0`, deci `a[0]` nu are valoarea implicită.

10.2.2 Implementarea (o variantă de nota șase)

Tabloul inițializat virtual cu elemente de tip `T`, `tablouVI<T>` (se poate citi chiar "tablou șase"), este o clasă cu o funcționalitate suficient de bine precizată pentru a nu pune probleme deosebite la implementare. Vom vedea ulterior că apar totuși anumite probleme, din păcate majore și imposibil de ocolit. Până atunci, să stabilim însă structura clasei `tablouVI<T>`. Fiind, în esență, vorba tot de un tablou, folosim clasa `tablou<T>` ca tip public de bază. Altfel spus, `tablouVI<T>` este un subtip al tipului `tablou<T>` și poate fi folosit oricând în locul acestuia. Alături de datele moștenite de la tipul de bază, noua clasă are nevoie de:

- Cele două tablouri auxiliare `p` și `b`.
- Întregul `sb`, contorul locațiilor ocupate din `b`.
- Elementul `vi`, în care vom memora valoarea implicită a elementelor tabloului.

În privința funcțiilor membre avem nevoie de:

- Un constructor (constructorii nu se moștenesc), pentru a dimensiona tablourile și a fixa valoarea implicită.
- O funcție (operator) de inițializare virtuală, prin care, în orice moment, să "inițializăm" tabloul.
- Un operator de indexare.

În mare, structura clasei `tablouVI<T>` este următoarea:

```
template <class T>
class tablouVI: public tablou<T> {
public:
  tablouVI( int, T );

  tablouVI& operator =( T );
  T& operator []( int );
```

```
private:
  T vi;             // valoarea implicita

  tablou<int> p, b; // tablourile auxiliare p si b
  int sb;           // pointer in b
};
```

unde operatorul de atribuire este cel care realizează inițializarea virtuală a tabloului.

Indexarea este operația cea mai dificil de implementat. Dificultatea provine din necesitatea de a-i conferi acestui operator o funcționalitate similară celui din clasa `tablou<T>`, în sensul de a putea fi folosit atât pentru returnarea valorii elementelor, cât și pentru modificarea lor. Pentru a nu complica în mod inutil implementarea, convenim ca primul acces la fiecare element să implice și inițializarea elementului respectiv cu valoarea implicită. În consecință, modificarea valorii unui element se realizează prin simpla returnare a referinței elementului. Operatorul de indexare este implementat astfel:

```
template<class T>
T& tablouVI<T>::operator []( int i) {
  static T z;   // elementul returnat in caz de eroare

  // verificarea indicelui i
  if ( i < 0 || i >= d ) {
    cerr << "\n\ntablouIV -- " << d
         << ": indice eronat: " << i << ".\n\n";
    return z;
  }

// returnarea valorii elementului i
  int k = p[ i ];
  if ( 0 <= k && k <= sb && b[ k ] == i )
    // element deja initializat
    return a[ i ];
  else
    // elementul se initializeaza cu valoarea implicita
    return a[ b[ p[ i ] = ++sb ] = i ] = vi;
}
```

Operatorul de atribuire implementat mai jos poate fi oricând invocat pentru inițializarea virtuală. Argumentul lui este valoarea implicită asociată tuturor elementelor tabloului:

```
template<class T>
tablouVI<T>& tablouVI<T>::operator =( T v ) {
  vi = v; sb = -1;
  return *this;
}
```

De asemenea, putem realiza inițializarea virtuală și prin intermediul constructorului clasei `tablouVI<T>`:

```
template<class T>
tablouVI<T>::tablouVI( int n, T v ):
            tablou<T>( n ), vi( v ), p( n ), b( n ), sb( -1 ) {
}
```

Operațiile de inițializare a obiectelor prin constructori constituie una din cele mai bine fundamentate părți ale limbajului C++. Pentru clasa `tablou<T>`, inițializarea elementelor tabloului este ascunsă în constructorul

```
template<class T>
tablou<T>::tablou( int dim ) {
  a = 0; v = 0; d = 0;      // valori implicite
  if ( dim > 0 )            // verificarea dimensiunii
    a = new T [ d = dim ];  // alocarea memoriei
}
```

constructor invocat prin lista de inițializare a membrilor din constructorul clasei `tablouVI<T>`. Mai exact, expresia `new T[ d ]` are ca efect invocarea constructorului implicit al tipului `T`, pentru fiecare din cele `d` elemente alocate dinamic. Acest comportament (absolut justificat) al operatorului `new` este total inadecvat unei inițializări în timp constant. Ne întrebăm dacă putem doar aloca spațiul, fără a-l și inițializa. Dacă tipul `T` nu are nici un constructor, atunci inițializarea spațiului alocat este inutilă, deoarece acest tip admite obiecte neinițializate. Dar, dacă tipul `T` are cel puțin un constructor, atunci înseamnă că obiectele de tip `T` nu pot fi neinițializate și, în consecință, este necesar un constructor implicit (apelabil fără nici un argument) pentru a inițializa spațiul alocat prin `new`. Astfel, am ajuns la primul motiv pentru care această implementare este doar de nota șase: tabloul `tablouVI<T>` este (virtual) inițializat în timp constant, numai dacă tipul `T` nu are constructor, altfel spus, dacă permite lucrul cu obiecte neinițializate.

Problema pe care ne-o punem acum este în ce măsură responsabilitatea verificării acestei condiții poate fi preluată de compilator sau de proiectantul clasei `tablouVI<T>`. Compilatorul poate semnala, în cel mai bun caz, absența constructorului implicit. Proiectantul nu este nici el într-o situație mai bună, deoarece:

- Nu poate modifica comportamentul operatorului `new` astfel încât să nu mai invoce constructorul implicit.
- Prezența (sau absența) constructorilor clasei `T` nu poate fi verificată în timpul rulării programului.

Soluția este reproiectarea clasei, pentru a se obține o variantă mai puțin naivă. De exemplu, în tabloul propiu-zis, se pot memora adresele elementelor, și nu elementele.

Obiectele de tip `tablouVI<T>` generează necazuri și în momentul în care încetează să mai existe. Știm că, în aceste situații, se vor invoca destructorii datelor membre și cel al clasei de bază (în această ordine). Ajungem din nou la clasa `tablou<T>` și la destructorul acesteia:

```
~tablou( ) { delete [ ] a; }
```

care invocă destructorul clasei `T` (în caz că `T` are destructor) pentru fiecare obiect alocat la adresa `a`. Efectele destructorului asupra obiectelor care nu au fost niciodată inițializate sunt greu de prevăzut. Rezultă că prezența destructorului în clasa `T` este chiar periculoasă, spre deosebire de prezența constructorului care va genera doar pierderea timpului constant de inițializare.

Continuând analiza deficiențelor clasei `tablouIV<T>`, ajungem la banala operație de atribuire `a[ i ] = vi` din operatorul de indexare. Dacă tipul `T` are un operator de atribuire, atunci acest operator consideră obiectul invocator (cel din membrul drept) deja inițializat și va încerca să-l distrugă în aceeași manieră în care procedează și destructorul. În cazul nostru, premisa este contrară: `a[ i ]` nu este inițializat, deci nu ne trebuie o operație de atribuire, ci una de inițializare a obiectului din locația `a[ i ]` cu valoarea `vi`. Iată un nou argument în favoarea utilizării unui tablou de adrese și nu a unui tablou de obiecte.

Fără a mai conta la nota acordată, să observăm că operațiile de inițializare și de atribuire între obiecte de tip `tablouVI<T>` sunt nu numai generatoare de surprize (neplăcute), ci și foarte ineficiente. Surprizele sunt datorate constructorilor și destructorilor clasei `T` și au fost analizate mai sus. Ineficiența se datorează faptului că nu este necesară parcurgerea în întregime a tablourilor implicate în transfer, ci doar parcurgerea elementelor purtătoare de informație (inițializate). Cauza acestor probleme este operarea membru cu membru în clasa `tablouVI<T>`, prin intermediul constructorului de copiere și al operatorului de atribuire din clasa `tablou<T>`.

Concluzia este că tabloul inițializat virtual generează o mulțime de probleme. Aceasta, deoarece procesul de inițializare și cel opus, de distrugere, sunt tocmai elementele imposibil de ocolit în semantica structurilor de tip clasă din limbajul C++. Implementarea prezentată, chiar dacă este doar de nota șase, poate fi

utilizată cu succes pentru tipuri de date predefinite, sau care nu necesită constructori și destructori. De asemenea, se vor evita operațiile de copiere (inițializări, atribuiri, transmiteri de parametri prin valoare) între obiectele de acest tip.

10.2.3 `tablouVI<T>` ca subtip al tipului `tablou<T>`

Derivarea publică instituie o relație specială între tipul de bază și cel derivat. Tipul derivat este un subtip al celui de bază, putând fi astfel folosit oriunde este folosit și tipul de bază. Această flexibilitate se bazează pe o conversie standard a limbajului C++, și anume conversia de la tipul derivat public către tipul de bază.

Prin funcționalitatea lui, tabloul inițializat virtual este o particularizare a tabloului. Decizia de a construi tipul `tablouVI<T>` ca subtip al tipului `tablou<T>` este deci justificată. Simpla derivare publică nu este suficientă pentru a crea o veritabilă relație tip-subtip. De exemplu, să considerăm următorul program pentru testarea clasei `tablouVI<T>`.

```
#include <iostream.h>
#include "tablouVI.h"

// declaratie necesara pentru a evita
// referirea la sablon - vezi Sectiunea 4.1.3
ostream& operator <<( ostream&, tablou<int>& );

main( ) {
  cout << "\nTablou (de intregi) initializat virtual."
       << "\nNumarul elementelor, valoarea implicita ... ";
  int n, v;  cin >> n >> v;
  tablouVI<int> x6( n, v );

  cout << "\nIndicele, valoarea (prin indicele -1 se\n"
       << " modifica valoarea implicita) <EOF>:\n...";
  while( cin >> n >> v ) {
    if ( n == -1 ) x6 = v; else x6[ n ] = v;
    cout << "...";
  }
  cin.clear( );

  cout << '\n' << x6 << '\n';
  return 1;
}
```

Acest program este corect, dar valorile afișate nu sunt cele care ar trebui să fie. Cauza este operatorul de indexare `[]` din `tablou<T>`, operator invocat în funcția

```
template <class T>
ostream& operator <<( ostream& os, tablou<T>& t ) {
  int n = t.size( );

  os << " [" << n << "]: ";
  for ( int i = 0; i < n; os << t[ i++ ] << ' ' );
  return os;
}
```

prin intermediul argumentului `t`. Noi dorim ca, atunci când `t` este de tip `tablouVI<T>`, operatorul de indexare `[]` invocat să fie cel din clasa `tablouVI<T>`. De fapt, ceea ce urmărim este o legare (selectare) dinamică, în timpul rulării programului, a operatorului `[]` de tipul actual al obiectului invocator. Putem obține acest lucru declarând `virtual` operatorul de indexare din clasa `tablou<T>`:

```
template <class T>
class tablou {
  // ..
public:
  // ...
  virtual T& operator []( int );
  // ...
};
```

O funcție declarată `virtual` într-o clasă de bază este o funcție a cărei implementare depinde de tip, în sensul că va fi reimplementată pentru unele din tipurile derivate. Atunci când se invocă printr-o referință sau pointer la clasa de bază, funcția virtuală permite selectarea variantei sale redefinite pentru tipul actual al obiectului invocator. În cazul nostru, operatorii de indexare au fost redefiniți în clasa derivată `tablouVI<T>`. Deci, prin declararea ca funcții virtuale în clasa de bază, se realizează legarea lor dinamică de tipul actual al obiectului invocator.

Moștenirea și legăturile dinamice sunt atributele necesare programării orientate pe obiect. Limbajul C++ suportă aceste facilități prin mecanismul de derivare al claselor și prin funcțiile virtuale. Un alt element util programării orientate pe obiect este obținerea de informații asupra claselor în timpul rulării programului (*RTTI* sau **R**un-**T**ime **T**ype **I**nformation). Iată o situație simplă, în care avem nevoie de *RTTI*. Fie următoarea funcție pentru interschimbarea a două tablouri:

```
template <class T>
void swap( tablou<T>& a, tablou<T>& b ) {
  tablou<T> tmp = a;
  a = b;
  b = tmp;
}
```

Pentru a o invoca, putem utiliza orice argumente de tip `tablou<T>` sau `tablouVI<T>`. Nu este însă logic să interschimbăm un `tablouVI<T>` cu un `tablou<T>`. Detectarea acestei situații (corectă din punct de vedere sintactic) se poate face numai în momentul rulării programului, prin *RTTI*. Limbajul C++ nu are facilități proprii pentru *RTTI*, dar permite implementarea lor prin mecanismul funcțiilor virtuale. Multe din bibliotecile C++ profesionale oferă facilități sofisticate de *RTTI*. Pentru exemplul de mai sus, am implementat o variantă primitivă de *RTTI*. Este vorba de introducerea funcțiilor virtuale `tip()` în clasele `tablou<T>` și `tablouVI<T>`, funcții care returnează codurile de identificare ale claselor respective.

```
template <class T>
class tablou {
  // ...
public:
  // ...
  virtual char tip( ) const { return 'T'; }
  // ...
};

template <class T>
class tablouVI: public tablou<T> {
public:
  // ...
  char tip( ) const { return 'V'; }
  // ...
};
```

Deci, vom introduce în funcția `swap( tablou<T>&, tablou<T>& )` secvența de test a tipurilor implicate:

```
template <class T>
void swap( tablou<T>& a, tablou<T>& b ) {
  if ( a.tip( ) != b.tip( ) )
    cerr << "\n\nswap -- tablouri de tipuri diferite.\n\n";
  else {
    tablou<T> tmp = a; a = b; b = tmp;
  }
}
```

Am reușit, astfel, să prevenim anumite operații corecte sintactic, dar imposibil de aplicat obiectelor din tipurile derivate.

Mecanismul *RTTI* trebuie folosit cu mult discernământ. Este mai bine să prevenim situații ca cea de mai sus, decât să le soluționăm prin “artificii” (de tip *RTTI*) care pot duce la pierderea generalității funcțiilor sau claselor respective.

10.3 Exerciții

10.1 Dacă toate elementele unui tablou inițializat virtual au fost modificate, atunci testarea stării fiecărui element prin tablourile `p` și `b` este inutilă. Mai mult, spațiul alocat tablourilor `p` și `b` poate fi eliberat.

Modificați operatorul de indexare al clasei `tablouVI<T>` astfel încât să trateze și situația de mai sus.

10.2 Operatorul de indexare al clasei `tablouVI<T>` inițializează fiecare element cu valoarea implicită, chiar la primul acces al elementului respectiv. Procedeul este oarecum ineficient, deoarece memorarea valorii unui element are sens doar dacă această valoare este diferită de valoarea implicită.

Completați clasa `tablouVI<T>` astfel încât să fie memorate efectiv doar valorile diferite de valoarea implicită.

10.3 Ceea ce diferențiază operatorul de indexare din clasa `tablouVI<T>` față de cel din clasa `tablou<T>` este, în cele din urmă, verificarea indicelui:

- în clasa `tablou<T>` este o vorba de o simplă încadrare între `0` și `d`
- în clasa `tablouVI<T>` este un algoritm care necesită verificarea corelațiilor dintre tablourile `p` și `b`.

Implementați precedura virtuală `check( int )` pentru verificarea indicelui în cele două clase și modificați operatorul de indexare din clasa `tablou<T>` astfel încât operatorul de indexare din clasa `tablouVI<T>` să nu mai fie necesar.

10.4 Implementați constructorul de copiere, operatorul de atribuire și funcția de redimensionare pentru obiecte de tip `tablouVI<T>`.

Epilog

De la înmulțirea “a la russe” până la grafurile **AND**/**OR** am parcurs de fapt o mică istorie a gândirii algoritmice. Am pornit de la regulile aritmetice din Antichitate și am ajuns la modelarea raționamentului uman prin inteligența artificială. Acestă evoluție spectaculoasă reflectă, de fapt, evoluția noastră ca ființe raționale.

S-ar putea ca pașii făcuți să fi fost uneori prea mari. La aceasta a dus dorința noastră de a acoperi o arie suficient de largă. Pe de altă parte, este și efectul obiectului studiat: eleganța acestor algoritmi impune o exprimare concisă. Mai mult, limbajul C este cunoscut ca un limbaj elegant, iar limbajul C++ accentuază această caracteristică. Interesant acest fenomen prin care limbajul ia forma obiectului pe care îl descrie. Cartea noastră este, în mod ideal, ea însăși un algoritm, sau un program C++.

Este acum momentul să dezvăluim obiectivul nostru secret: am urmărit ca, la un anumit nivel, implementarea să fie cât mai apropiată de pseudo-cod. Detaliile legate de programarea orientată pe obiect devin, în acest caz, neimportante, utilizarea obiectelor fiind tot atât de simplă ca invocarea unor funcții de bibliotecă. Pentru a ajunge la această simplitate este necesar ca cineva să construiască bine clasele respective. Cartea noastră reprezintă un prim ghid pentru acel “cineva”.

Nu putem încheia decât amintind cuvintele lui Wiston Churchill referitoare la bătălia pentru Egipt:

Acesta nu este sfârșitul.
Nu este nici măcar începutul.
Dar este, poate, sfârșitul
începutului.

Bibliografie selectivă

Brassard, G., Bratley, P. *"Algorithmics – Theory and Practice"*, Prentice-Hall, Englewood Cliffs, 1988.

Cormen, T.H., Leiserson, C.E., Rivest, R.L. *"Introduction to Algorithms"*, The MIT Press, Cambridge, Massachusetts, 1992 (eighth printing).

Ellis, M., Stroustrup, B. *"The Annotated C++ Reference Manual"*, Addison-Wesley, Reading, 1991.

Graham, R.L., Knuth, D.E., Patashnik, O. *"Concrete Mathematics"*, Addison-Wesley, Reading, 1989.

Horowitz, E., Sahni, S. *"Fundamentals of Computer Algorithms"*, Computer Science Press, Rockville, 1978.

Knuth, D.E. *"Tratat de programarea calculatoarelor. Algoritmi fundamentali"*, Editura Tehnică, București, 1974.

Knuth, D.E. *"Tratat de programarea calculatoarelor. Sortare și căutare"*, Editura Tehnică, București, 1976.

Lippman, S. B. *"C++ Primer"*, Addison-Wesley, Reading, 1989.

Livovschi, L., Georgescu, H. *"Sinteza și analiza algoritmilor"*, Editura Științifică și Enciclopedică, București, 1986.

Sedgewick, R. *"Algorithms"*, Addison-Wesley, Reading, 1988.

Sedgewick, R. *"Algorithms in C"*, Addison-Wesley, Reading, 1990.

Sethi, R. *"Programming Languages. Concepts and Constructs"*, Addison-Wesley, Reading, 1989.

Smith, J.H. *"Design and Analysis of Algorithms"*, PWS-KENT Publishing Company, Boston, 1989.

Standish, T.A. *"Data Structure Techniques"*, Addison-Wesley, Reading, 1979.

Stroustrup, B. *"The C++ Programming Language"*, Addison-Wesley, Reading, 1991.

Stroustrup, B. *"The Design and Evolution of C++"*, Addison-Wesley, Reading, 1994.