

Tema „Analiza algoritmilor TEHNICII PROGRAMĂRII”

1. Importanța implementării algoritmilor eficienți

1.1. Importanța studiului algoritmilor. În rezolvarea unei probleme cu calculatorul este importanta definirea strategiei de rezolvare a problemei și definirea optimă a algoritmului. Există câteva metode speciale de elaborare a algoritmilor. Alegerea uneia sau alteia fiind făcută în concordanță cu specificul problemei de rezolvat și cu eficiența algoritmului. Fiecare algoritm are un **context în care este folosit**. Metodele (strategiile, tehnicile de rezolvare) care vor fi luate în discuție sunt: *Metodele euristice*, metoda *Backtracking*, metoda *"Greedy"*, metoda *"divide et impera"*, metoda *programării dinamice* și metoda *Branch and Bound*.

Dacă să ne amintim din definiția algoritmului care 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. Iar *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. Având în vedere că ați scris deja programe în C, probabil că ați avut uneori dificultăți în a formula soluția pentru o problemă. Alteori, poate că nu ați putut decide care dintre algoritmii ce rezolvau aceeași problemă este mai bun. Aceasta temă 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 (strategie, reguli) + creativitate (cunoștințe, intuiție) = soluție eficientă.

Un obiectiv major al acestei compartiment 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.

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 prezenta 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).

Pentru fiecare algoritm există un *domeniu de definiție* al cazurilor pentru care algoritmul funcționează corect. Orice calculator limitează mărimea cazurilor cu care poate opera. Aceasta limitare nu poate fi însă atribuită algoritmului respectiv. Încă o dată, observăm că există o diferență esențială între programe și algoritmi.

1.2. Metodele euristice Euristicele sunt tehnici care ajută la descoperirea unei soluții, deși nu prezintă garanția că aceasta e soluția optimă. Soluțiile euristice se bazează pe metode de rezolvare rezultate mai mult din experiență practică, care încearcă să estimeze o soluție acceptabilă, eficientă din punct de vedere computațional, cât mai apropiată de soluția optimă.

Euristică- metodă de studiu și de cercetare bazată pe descoperirea de fapte noi. Gândirea euristică nu se produce direct, implică judecata, căutarea, iar cunoașterea obținută poate fi modificată în vederea soluționării problemei.

Prin *metode euristice* vom înțelege acele metode care:

- oferă soluții acceptabile, nu neapărat optimale.
- se pot implementa relativ ușor, conducând la rezultate într-un timp rezonabil, de obicei mult mai mic decât prin aplicarea metodelor exacte.

Există euristici specifice unor anumite situații particulare și euristici generale, aplicabile pentru o gamă largă de probleme

1.2.1. Principii care stau la baza implementării metodelor euristice:

▪ **Descompunerea** problemei în mai multe subprobleme, analog ca la metoda Greedy. Nu putem fi siguri că din combinarea optimelor locale rezultă optimul global. Problemele se pot simplifica dacă se reduc anumite condiții ale problemei greu de îndeplinit.

▪ **Căutarea** mai multor soluții aproximative din care se alege cea mai bună, adică aceea care dă diferențe neglijabile față de rezultatul optim.

Ca exemplificare a metodei se consideră **problema comis-voiajorului**: Fiind date n puncte de vizitare și distanțele între aceste puncte, se cere să se determine *traseul de lungime minimă* pe care trebuie să-l parcurgă comis-voiajorul pentru a vizita toate aceste puncte și să se întoarcă de unde a plecat.

Pentru rezolvare se modelează punctele de vizitare cu nodurile unui graf, iar lungimea unui traseu de la nodul i la nodul j cu un arc în graf. Arcului i se asociază lungimea l_{ij} . Lungimea se memorează în elementul a_{ij} al matricei a .

În abordarea euristică a acestui caz, comis-voiajorul va alege totdeauna cea mai apropiată localitate nevizitată. Desigur, această strategie nu conduce la un optim, dar considerând ca punct de plecare orice punct din rețea, pentru fiecare punct se va determina traseul minim.

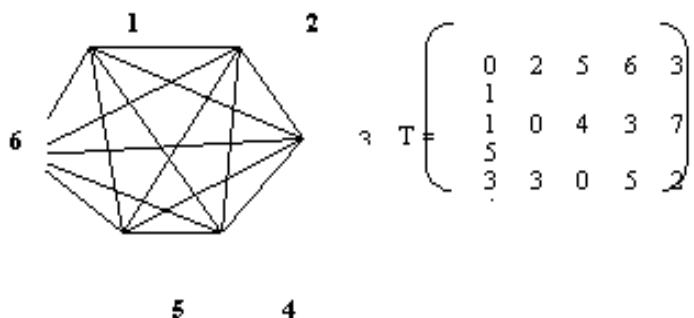
i. Algoritmi euristici

Orice metodă nefundamentată teoretic, care conduce repede la o soluție, aplicabilă și în condiții de lucru fără calculator este un **algoritm euristic**.

De remarcat că și metodele exacte sau aproximative se bazează pe o euristică, dar la care s-a putut atașa și o fundamentare teoretică. O euristică mult răspândită este și metoda Greedy, dar în cazul structurilor matroidale este fundamentată teoretic. Situație întâlnită rar în practică.

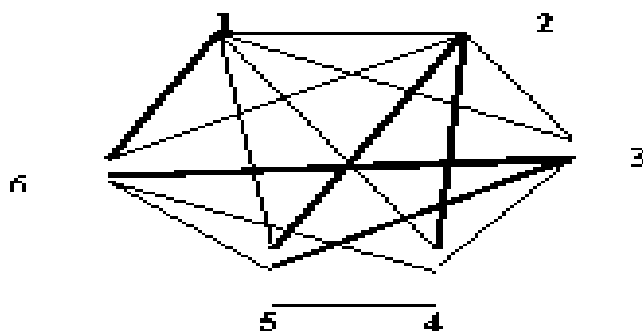
Ordonantare pe o mașină a operațiilor

Metoda euristică



Matricea timpilor de trecere de la o operație la alta este T .

Pornim de la produsul 1 (nodul 1) elementul minim pe linia 1 este $t_{16} = 1$ marcăm arcul 1-6



În linia 6 elementul minim este t_{51} dar nu avem voie să facem ciclu numai când au fost parcurse toate produsele. Deci alegem minimul următor $t_{63} = 3$ (Se putea alege $t_{65} = 3$) marcăm arcul 6-3.

În linia 3 avem elementul minim $t_{35} = 2$, din linia 5 avem elementul minim t_{52} sau $t_{54} = 2$. Marcăm arcurile 3-5 și 5-2.

La sfârșit în linia 2 minimul este atins de $t_{21} = 1$ dar nu am vizitat încă nodul 4 așa că alegem elementul $t_{24} = 4$.

Avem ordonarea produselor 1-6-3-5-2-4 iar timpii de pregătire totală este $= 1 + 3 + 2 + 2 + 3 = 11$.

Se poate observa că plecând dintr-un alt nod putem obține alte soluții mai bune sau mai puțin bune.

ii. Sistemele bazate pe cunoștințe

Entuziasmul care a înconjurat cercetările în domeniul inteligenței artificiale în anii de pionierat se pare că a dispărut în ultima vreme.

E vorba de un eșec răsunător sau de un succes discret?

Sistemele bazate pe cunoștințe reprezintă o tehnologie modernă dar controversată din punct de vedere al aplicabilității ei practice. Pe de o parte, domeniul este criticat deoarece realizările efective par a nu fi la nivelul speranțelor anilor '70-'80. Pe de altă parte, deși sistemele actuale sunt mai puțin ambițioase decât cele visate cu 15 ani în urmă, ele există și sunt funcționale, asistând specialiștii, oferind expertiză prețioasă și realizând economii semnificative companiilor care le utilizează.

Acest material prezintă principiile și arhitectura sistemelor bazate pe cunoștințe, specificând coordonatele de bază ale tehnologiei sistemelor bazate pe cunoștințe, cât și locul acestei noi tehnologii în știința calculatoarelor.

Una dintre cele mai dificile probleme ale reprezentării cunoștințelor într-un sistem inteligent este reprezentarea **cunoștințelor euristice**. Cunoștințele euristice reprezintă o formă particulară de cunoștințe utilizată de oameni pentru a rezolva probleme complexe. Ele reprezintă cunoștințele utilizate pentru a judeca corect, pentru a lua o decizie, pentru a avea un comportament după o anumită strategie sau a utiliza trucuri sau reguli de bun simț. Acest tip de cunoștințe nu este nici formalizat, nici demonstrat a fi eficient și câteodată nici corect, dar este frecvent utilizat de oameni în rezolvarea problemelor.

Utilizarea cunoștințelor în sistemele bazate pe cunoștințe necesită o formă de raționament, decizii și acțiuni în scopul obținerii de noi cunoștințe care în final vor reprezenta soluția problemei de rezolvat. Aceste acțiuni se fac cu ajutorul motorului de inferență.

Aceste tipuri de probleme sunt foarte complexe și necesită instrumente de abordare adecvate: programare logică, metode de programare euristice care să furnizeze soluții bune (chiar dacă nu optime) într-un timp scurt; tehnicile enumerate, care permit găsirea soluției într-un spațiu de căutare de dimensiuni foarte mari, sunt dezvoltate tot în cadrul inteligenței artificiale.

Cunoștințele euristice reprezintă o formă particulară de cunoștințe utilizată de oameni pentru a rezolva probleme complexe. Ele reprezintă cunoștințele utilizate pentru a judeca corect, pentru a lua o decizie, pentru a avea un comportament după o anumită strategie sau a utiliza trucuri sau reguli de bun simț. Acest tip de cunoștințe nu este nici formalizat, nici demonstrat a fi eficient și câteodată nici corect, dar este frecvent utilizat de oameni în numeroase situații. Judea Pearl, în lucrarea sa "*Heuristics. Intelligent Search Strategies for Computer Problem Solving*", definește cunoștințele euristice astfel:

"Euristicele sunt criterii, metode sau principii pentru a alege între diverse alternative de acțiune pe aceea care promite a fi cea mai eficientă în realizarea unui scop. Ele reprezintă compromisuri între două cerințe: necesitatea de a lucra cu criterii simple și, în același timp, dorința de a vedea că aceste criterii fac o selecție corectă între alternativele bune și rele."

Heuristic Search Strategies = Strategii de căutare euristice

Informed Search Strategies = Strategii de căutare informate

Strategiile de căutare *informate*, numite și *strategii euristice*, consideră stările următoare de inspectat pe baza unor funcții de evaluare sau după criterii euristice. Strategia folosește o funcție de evaluare a situației globale sau locale care indică starea următoare cea mai promițătoare din punct de vedere al avansului spre soluție.

1.2.2. Raționamente Euristice. Un aspect esențial al componentei operative a gândirii – *strategiile rezolutive* – se releva în procesul rezolvării de probleme. Activitatea gândirii este solicitată în mod esențial de probleme, care pot avea grade de dificultate diferite, după cum pot aparține unor tipuri foarte variate. În termeni psihologici, o problema se definește ca un obstacol sau o dificultate cognitivă care implică o necunoscută (sau mai multe) și fata de care repertoriul de răspunsuri câștigat în experiența anterioară apare insuficient sau inadecvat.

Rezolvarea problemei înseamnă depășirea obstacolului / dificultății, recombinașd datele experienței anterioare în funcție de cerințele problemei. O. Selz și M. Wertheimer considera problematica – o situație ce prezintă o „lacuna acoperită”, un element criptic, iar M. Mager releva caracterul de „situație deschisă”, generatoare de tensiune psihică, odată cu nevoia de „închidere” a structurii incomplete. O situație problematică presupune un conflict cognitiv creat de raportul dintre cunoscut și necunoscut, o „disonanță” internă iscată de decalajul între resurse actuale și cerințe, rezolvarea însăși impunând tatonări repetate, deci un efort de voință. În sensul arătat, constituie probleme nu numai cele din domeniul matematicii sau fizicii, dar și din domeniul tehnic, precum și din oricare altul. De pildă, determinarea unei plante (la botanică), analiza sintactică a unei fraze (la gramatică), un comentariu de ordin stilistic (la literatură) etc., reprezintă la rândul lor, probleme pentru că impun depășirea cognitivă a unui obstacol sau a unei dificultăți.

Ca metode de investigare a procesului rezolutiv se utilizează: tehnica „gândirii cu voce tare”, înregistrarea mișcărilor oculare, consemnarea altor comportamente sau indicatori (de exemplu, reacțiile electrodermale) ș.a. Tehnica „gândirii cu voce tare” – cunoscută de peste șase decenii în psihologie – impune subiectului să dezvăluie în cuvinte mersul gândirii, intențiile ce se conturează, ipotezele care apar pe parcurs, deci întreg conținutul conștiinței legat de rezolvarea problemei, ceea ce se consemnează într-un protocol fidel. Pe baza acestor relații verbale se reconstituie apoi pas cu pas, pe unități sau secvențe determinate întreg procesul dezvoltării.

Desigur, procesul căutării (în limbaj interior) se desfășoară rapid, astfel încât nu toate detaliile sale ajung să se reflecte în verbalizările subiectului. Suprapunerea protocoalelor poate oferi totuși o imagine satisfăcătoare asupra procesului gândirii. Practic, protocolul este descompus în fraze scurte, care se etichetează și numerotează. Frazarea se bazează pe o evaluare a ceea ce constituie un episod, o referință, un fapt. Procesul rezolvării se poate reda sub forma unui graf cu arborescențe, numit *graf-arbore*, care descompune demersul respectiv în etape și pași mai mici, arătându-se ramificațiile urmate.

Înregistrarea mișcărilor oculare, mai exact a traseelor oculare (sacade, zone de fixare a atenției etc.), poate dubla relatarea verbală a subiectului, astfel ca din sincronizarea acestor două feluri de informații să se poată surprinde mai bine procesul de rezolvare. Înregistrarea mișcărilor vizuale poate suplimenta deci comportamentul verbal, datele obținute se vor suprapune sau completa reciproc, exteriorizând procesul rezolutiv în vederea unei analize experimentale.

Inițial, o asemenea metodă combinată s-a utilizat în studierea jocului de șah pentru a urmări dinamica activității de explorare-căutare înaintea efectuării unei mișcări. În paralel, s-au studiat și șahiștii orbi, care se bazează pe simțul tactil-kinestezic pentru examinarea pozițiilor pe tabla de șah, exteriorizând astfel în mai mare măsură pe plan motric procesul gândirii.

În rezolvarea de probleme alternează de regula, *strategii sistematice* – *uneori algoritmice* – și *strategii euristice*. Strategiile algoritmice cuprind scheme de lucru fixate în prescripții precise, care pot fi învățate, asigurând obținerea certă a rezultatului.

În definiție exactă, algoritmul este o prescripție precisă ce nu lasă loc arbitrarului, prescripție care permite ca, plecând de la date inițiale – variabile în anumite limite – să se ajungă la rezultatul căutat (A.A.

Markov). Trei note apar ca fiind definitorii pentru ceea ce se numește algoritm: caracterul precis determinat, valabilitatea sa pentru o clasa întreaga de probleme și finalitatea certa.

Daca ne gândim, de exemplu, la rezolvarea unei ecuații de gradul II, de forma completa, aceasta urmează o schema precisă data de formula.

În care se arată suita de operații necesare pentru a ajunge la rezultat. Fiecare simbol indică o acțiune: -b înseamnă a lua coeficientul termenului de gradul întâi cu semn schimbat, b^2 indică mulțimea lui b cu el însuși ș.a.m.d. Exemplul dat ilustrează ceea ce se poate numi o prescripție algoritmică. Ea este completă, analitică și avansează secvențial.

În mod analog se pot propune prescripții de tip algoritmic pentru analiza sistematică a unei fraze sau pentru recunoașterea unei planete, nota algoritmică formează doar *canavaua schematica* a activității. Într-o schematizare grafică întreaga desfășurare poate fi redată printr-un „arbore” cu ramificații dihotomice care comportă deci, la fiecare nod, decizii binare.

În fața unei probleme noi sau complexe, pentru care nu se cunosc încă proceduri tipice, rezolvitorul nu se mai poate baza pe un set de reguli (algoritm) care să-i garanteze obținerea soluției. Teoretic, el se va afla în fața unui număr mare de alternative posibile, care nu pot fi triate toate, astfel încât se impune utilizarea unor *strategii euristice*. H. Simon, laureat al premiului Nobel, arată că rezolvarea de probleme este caracterizată, teoretic, ca un proces de căutare și parcurgere de la un capăt la altul a unui *arbore* poate, mai exact, a unui *graf orientat*), a cărui noduri reprezintă stări de fapt sau situații și ale cărui ramuri sunt operații care transformă o situație în alta. Graful conține un nod de plecare și unul sau mai multe noduri-scop. A rezolva o problemă înseamnă a găsi o secvență de operații care transformă situația de plecare în situație-scop, adică un drum de la nodul de start la nodul-scop. Succesul unui rezolvitor de probleme constă în capacitatea de a decupa pentru investigare doar o mică parte din ansamblul de posibilități (alternative) pe care-l comportă teoretic problema, decupare în măsura să ducă totuși la rezultatul corect.

Această selecție are loc prin *procedee euristice*, raționamente neformalizate care urmează scheme fluente. Raționamentul euristic este, prin excelență, de natură probabilistă, dar teoria probabilităților se aplică aici numai calitativ (G. Polya).

Pentru exemplificare, să ne gândim la jocul de șah, mai exact la un moment al unei partide, când pe tabla de șah s-ar afla numai 10 figuri albe și 10 negre, prezentând fiecare posibilitatea (în medie) a câte 6 mutări. Se estimează că pentru a găsi cele 2 mutări optime următoare ar trebui să se cerceteze 6^{40} posibilități, ceea ce evident ar depăși capacitatea unui subiect uman. Un calculator ar putea efectua o asemenea operație într-un număr foarte mare de ani. Este necesar deci să intervină o alegere euristica a strategiilor, jucătorul nu poate explora practic toate posibilitățile (alternativele).

Din experiența îndelungată a jocului de șah – în primul rând a marilor maeștri – se extrag moduri de abordare euristica, ce pot fi introduse și în programul calculatorului. Asemenea abordări ar fi: căutați să obțineți controlul centrului înainte de a ataca asigurați regele, nu scoateți dama în joc prea devreme, dezvoltați caii înaintea nebunilor etc. Constituie prescripție sau regula euristica orice principiu sau procedeu care reduce sensibil activitatea de căutare a soluției. Firește, asemenea reguli nu garantează soluția – așa cum se întâmplă în strategiile algoritmice – dar pot duce în multe cazuri la rezolvare în mod economic și cu o anumită flexibilitate.

Analiza mijloace-scop (means-ends analysis) a fost recomandată de Simon și Newell. Ea pornește de la descompunerea problemei în starea inițială (So) (=datele problemei) și starea finală (Sf) (soluția problemei sau numele acestei soluții). Rezolvarea problemei constă în detectarea diferențelor dintre cele două stări și reducerea succesivă a acestor diferențe pe baza unor reguli până la anularea lor. Această euristica, formalizată și implementată a constituit programul G.P.S (General Problem Solving) care a demonstrat în chip original unele din teoremele logicii matematice.

Analiza prin sinteză este procesul prin care obiectul, în procesul gândirii, este inclus în relații noi, gratie cărora i se conferă proprietăți noi, exprimabile în noțiuni noi, dezvăluindu-se astfel un conținut informațional nou. Avem de-a face cu o alternanță foarte rapidă între percepție și gândire; procesul de rezolvare se produce mai curând simultan, la niveluri diferite – (senzorial și logic-noțional) – fapt care are drept rezultanta exterioară reformularea continuă a problemei. De aici și remarcă generală a psihologului amintit examinând o problemă oarecare o reformulăm și reformulând-o o rezolvăm, astfel încât procesul rezolvării ne apare – în expresie exterioară – ca un șir de reformulări.

Din punct de vedere logic am spune că informațiile obținute în procesul rezolvării își schimbă funcția, din indicativă în imperativă. Revenind prescriptive, aceste conținuturi informaționale orientează mersul ulterior al rezolvării problemei, determinând pe rezolvitor să aleagă anumite ramuri ale arborelui rezolutiv și nu altele.

Utilizarea euristicilor în procesul rezolutiv sta la baza dihotomiei experți-novici. Datele furnizate de psihologia rezolvării de probleme dovedesc ca problemele sunt rezolvate diferit de experți fata de novici. La rezolvarea problemelor de fizica de pilda exista doua deosebiri importante: a) experții își organizează cunoștințele în unități semnificative; novicii procedează pas cu pas; b) experții rezolva problemele pornind de la cunoscut la necunoscut; novicii pornesc de la variabila recunoscuta, fac apel la ecuația în care ea apare și pe baza ei încearcă sa calculeze acesta variabila (deci pornesc de la necunoscut spre cunoscut).

1.2.3. Tehnici-solving bazate pe euristici în timp real

Problema comis voiajorului Un *comis-voiajor* are de vizitat un număr de orașe, la sfârșitul voiajului el întorcându-se în localitatea de plecare. Deplasarea dintr-o localitate în alta presupune anumite *cheltuieli* (sau *distanțe* de parcurs sau *durate* de mers).

În ce ordine trebuie vizitate orașele astfel încât costul total al deplasării (sau lungimea traseului sau durata călătoriei) să fie minim?

Astfel enunțată, problema comis-voiajorului (notată pe scurt TSP de la Travelling Salesman Problem) este una din cele mai grele probleme de optimizare combinatorială. Pe lângă importanța teoretică, ea are numeroase aplicații practice. De exemplu, stabilirea celor mai "economice" trasee turistice într-un mare oraș sau zonă istorică (din punctul de vedere al costurilor sau distanțelor) este o problemă a comis-voiajorului. De asemenea, stabilirea ordinii în care un număr de operații (joburi) vor fi executate pe un utilaj, astfel încât suma timpilor de pregătire ai utilajului să fie cât mai mică, este o problemă de același tip.

Modelul matematic al problemei comis voiajorului Să notăm cu $0, 1, \dots, n$ orașele problemei, 0 fiind orașul din care comis-voiajorul pleacă și în care revine la terminarea călătoriei. Fie c_{ij} *costul* deplasării din localitatea i în localitatea $j \neq i$; punem $c_{ij} = \infty$ dacă nu există legătură directă între i și j sau în cazul $i = j$. Un *traseu* va fi descris cu ajutorul variabilelor *bivalente* :



Urmează a fi minimizată funcția:

$$f = \sum_{i,j=0}^n c_{ij} x_{ij} \quad (2.1)$$

Din orice oraș i , comis-voiajorul trebuie să se îndrepte către o altă localitate, astfel că:

$$\sum_{j=0}^n x_{ij} = 1 \quad i = 0, 1, \dots, n \quad (2.2)$$

În orice oraș ar ajunge, comis-voiajorul vine dintr-o localitate anterior vizitată, așa că:

$$\sum_{i=0}^n x_{ij} = 1 \quad j = 0, 1, \dots, n \quad (2.3)$$

Observăm că ansamblul (2.1) - (2.3) constituie modelul matematic (PA) al unei probleme generale de afectare. O examinare mai atentă a problemei arată că restricțiile (2.2) și (2.3) nu definesc numai trasee ce trec prin toate orașele (o singură dată!) ci și "reuniuni" de circuite disjuncte mai mici! Astfel, într-o problemă cu 7 orașe $0, 1, \dots, 6$ ansamblul:

$x_{03} = x_{35} = x_{50} = x_{21} = x_{14} = x_{46} = x_{67} = x_{72} = 1$, $x_{ij} = 0$ în rest satisface restricțiile (2.2) și (2.3) dar nu constituie un traseu complet așa cum se vede din figura 2.1 Fig. 2.1

Pentru evitarea acestui neajuns asociem orașelor $1, 2, \dots, n$, diferite de localitatea de plecare și sosire 0 , variabilele y_1, y_2, \dots, y_n și introducem inegalitățile:

$$y_i - y_j + n \cdot x_{ij} \leq n - 1 \quad i, j = 1, 2, \dots, n ; i \neq j \quad (2.4)$$

În principiu noile variabile pot lua orice valoare reală. Vom arăta că dacă $x = (x_{ij})$ determină un traseu complet atunci există valori numerice y_1, y_2, \dots, y_n care împreună cu x_{ij} satisfac relațiile (4.1.4). Într-adevăr, definim y_i ca fiind numărul de ordine al localității i în cadrul traseului determinat de x . Pentru $i \neq j$ două situații sunt posibile:

- localitatea j nu urmează imediat după localitatea i . Atunci $x_{ij} = 0$ și (4.1.4) devine $y_i - y_j \leq n - 1$ ceea ce este adevărat având în vedere semnificația acordată valorilor y_1, y_2, \dots, y_n .
- localitatea j urmează imediat după localitatea i . Atunci $y_j = y_i + 1$, $x_{ij} = 1$ și (4.1.4) se verifică cu egalitate.

Să presupunem acum că $z = (x_{ij})$ definește o reuniune de "subtrasee" disjuncte. Alegem unul care nu trece prin localitatea 0 și fie k numărul deplasărilor cel compun. Presupunând că ar exista valori numerice y_1, y_2, \dots, y_n care să satisfacă (4.1.4) însumăm pe acelea corespunzătoare deplasărilor de la un oraș la altul din subtraseul ales. Obținem $n \cdot k \leq (n - 1) \cdot k$ ceea ce este evident fals.

În concluzie, ansamblul (2.1) - (2.4) constituie modelul "corect" al problemei comis-voiajorului. Este vorba de un *program mixt* ce utilizează atât variabile întregi (bivalente) cât și variabile care pot lua orice valoare. În continuarea unei observații anterioare, putem spune că *problema comis-voiajorului* (TSP) este o *problemă de afectare* (PA) (ansamblul (2.1) - (2.3)) cu restricțiile *speciale* (2.4).

În considerațiile de mai sus costurile c_{ij} nu sunt "*simetrice*" adică este posibil ca $c_{ij} \neq c_{ji}$. *Problema simetrică a comis-voiajorului* se caracterizează prin $c_{ij} = c_{ji}$; aceasta se întâmplă dacă, de exemplu, c_{ij} sunt distanțe sau timpi de parcurgere.

Un caz particular al problemei simetrice este așa numita *problemă euclidiană a comis-voiajorului* în care distanțele c_{ij} satisfac inegalitatea triunghiului :

$$c_{ik} \leq c_{ij} + c_{jk} \quad (\forall) i, j, k.$$

Un algoritm exact de rezolvare a problemei comis voiajorului Următorul algoritm, de tip B&B, reduce rezolvarea TSP la rezolvarea unei *liste de probleme de afectare*. El s-a dovedit suficient de robust pentru probleme *asimetrice* cu un număr *moderat* de orașe. Principalul dezavantaj îl reprezintă faptul că numărul problemelor de afectare de rezolvat este imprevizibil și poate fi imens în cazul situațiilor reale. Algoritmul este datorat lui Eastman. Ca orice procedură de tip B&B, el utilizează:

- o "locație" x_{CMB} care va reține "cel mai bun" traseu găsit de algoritm până la un moment dat;
- o variabilă z_{CMB} care reține costul traseului memorat în x_{CMB} ;
- o listă L de probleme de afectare asemănătoare problemei (PA) și care diferă de problema (PA) inițială prin anumite costuri c_{ij} schimbate în ∞ ;

La start:

- x_{CMB} poate fi locația "vidă" în care caz $z_{CMB} = \infty$ sau reține un traseu arbitrar ales sau generat printr-o metodă euristică, x_{CMB} fiind inițializat cu costul aferent acestui traseu;
- lista L cuprinde numai problema (PA) inițială, determinată de (2.1) - (2.3);

Pasul 1. Dacă lista L este vidă **Stop**. Altminteri, selectează o problemă din lista L . Problema aleasă se șterge din L . Se trece la:

Pasul 2. Se rezolvă problema selectată. Dacă valoarea funcției obiectiv este $\geq z_{CMB}$ se revine la pasul 1. Altminteri, se trece la:

Pasul 3. Dacă soluția optimă este un traseu complet se actualizează x_{CMB} și z_{CMB} după care se revine la pasul 1. Altminteri se trece la:

Pasul 4. (Soluția optimă nu este un traseu ci o reuniune de "subtrasee mai mici"). Din soluția optimă se selectează circuitul cu cel mai mic număr de arce. Pentru fiecare arc (i, j) din circuitul ales (pentru care $x_{ij} = 1$) se adaugă la lista L problema obținută din cea rezolvată punând $c_{ij} = \infty$. Se revine la pasul 1.

Observații: 1) Rațiunea pasului 4 este clară: dacă $i \rightarrow j \rightarrow k \rightarrow \dots \rightarrow l \rightarrow i$ este circuitul selectat este clar că în traseul optim vom avea:

$$x_{ij} = 0 \text{ sau } x_{jk} = 0 \text{ sau } \dots \text{ sau } x_{li} = 0.$$

În consecință, vom căuta traseul optimal printre cele în care $x_{ij} = 0$ și dacă nu este acolo printre cele în care $x_{jk} = 0$ ș.a.m.d. Limitarea la traseele în care $x_{ij} = 0$ se face efectuând schimbarea $c_{ij} = \infty$.

2) La pasul 4, în lista L vor apare atâtea probleme câte arce are circuitul selectat; de aceea se alege circuitul cu cel mai mic număr de arce!

În raport cu termenii generali ai unei metode B&B, *ramificarea* are loc la pasul 4 în timp ce *mărginirea* se efectuează la pasul 2.

Exemplu Se consideră problema asimetrică a comis voiajorului cu cinci orașe, definită de următoarea matrice a costurilor:

| Orașe | 0 | 1 | 2 | 3 | 4 |
|-------|----------|----------|----------|----------|----------|
| 0 | ∞ | 10 | 25 | 25 | 10 |
| 1 | 1 | ∞ | 10 | 15 | 2 |
| 2 | 8 | 9 | ∞ | 20 | 10 |
| 3 | 14 | 10 | 24 | ∞ | 15 |
| 4 | 10 | 8 | 25 | 27 | ∞ |

Tabelul 2.1

Start. Se pleacă cu traseul $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 0$, care se reține în x_{CMB} și al cărui cost este $z_{CMB} = 10 + 10 + 20 + 15 + 10 = 65$. Lista L a problemelor de afectare ce urmează a fi rezolvate cuprinde numai problema inițială (PA).

Iterația 1. Se șterge (PA) din lista L și se rezolvă. Se obține soluția optimă:

$$x_{04} = x_{40} = x_{12} = x_{23} = x_{31} = 1, \quad x_{ij} = 0 \text{ în rest}$$

ce corespunde reuniunii următoarelor subtrasee:

$$0 \rightarrow 4 \rightarrow 0 \text{ și } 1 \rightarrow 2 \rightarrow 3 \rightarrow 1$$

Selectăm subtraseul $0 \rightarrow 4 \rightarrow 0$ și adăugăm la L problemele PA_1 și PA_2 derivate din PA înlocuind c_{04} respectiv c_{40} cu ∞ .

Iterația 2. Selectăm problema PA_1 și o rezolvăm. Lista L va cuprinde mai departe numai problema PA_2 . Pentru PA_1 se găsește soluția optimă:

$$x_{03} = x_{12} = x_{24} = x_{31} = x_{40} = 1, \quad x_{ij} = 0 \text{ în rest.}$$

care corespunde traseului complet $0 \rightarrow 3 \rightarrow 1 \rightarrow 2 \rightarrow 4 \rightarrow 0$. Costul său este $65 = z_{CMB}$. Nu am găsit deci un traseu mai bun decât cel pe care-l avem în x_{CMB} .

Iterația 3. Rezolvăm problema PA_2 , singura rămasă în L. Se obține soluția optimă:

$$x_{04} = x_{12} = x_{23} = x_{30} = x_{41} = 1, \quad x_{ij} = 0 \text{ în rest}$$

care corespunde traseului $0 \rightarrow 4 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 0$. Costul noului traseu este $62 < z_{CMB}$, drept care îl reținem în x_{CMB} și actualizăm $z_{CMB} = 62$.

Deoarece lista L este acum vidă traseul reținut în x_{CMB} este optim.

Tehnici euristice de rezolvare aproximativă a problemei comis voiajorului De obicei, soluția optimă a unei TSP este foarte greu de găsit (dacă nu chiar imposibil de găsit) atunci când numărul orașelor de vizitat este mare (peste 50). Pentru determinarea unei soluții măcar acceptabile au fost elaborate mai multe procedee euristice. Aceste procedee merită atenție din două puncte de vedere:

- se poate da un "*certificat de garanție*" pentru soluția obținută în sensul că este posibilă evaluarea "*depărtării*" maxime de soluția optimă;
- soluția aproximativă este găsită cu un *efort de calcul relativ moderat* și într-un *timp rezonabil*, aceasta însemnând că cei doi parametri depind *polinomial* de dimensiunea problemei (adică numărul de orașe);

În multe situații practice, procedeele euristice au condus chiar la soluția optimă dar acest lucru a fost confirmat numai prin aplicarea unui algoritm exact.

În principiu, o metodă euristică construiește o soluție *prin tatonare, din aproape în aproape, făcând la fiecare pas, cea mai bună alegere posibilă*. Din nefericire, această "schemă" nu conduce, de regulă, la cea mai bună alegere globală.

În continuare, vom prezenta mai multe euristici pentru rezolvarea *problemei euclidiene a comis voiajorului*, adică a problemei în care c_{ij} sunt *distanțe* ce satisfac:

- condiția de simetrie: $c_{ij} = c_{ji}$;
- inegalitatea triunghiului $c_{ik} \leq c_{ij} + c_{jk}$;

Euristica E_1 (mergi la cel mai apropiat vecin)

- se pleacă din localitatea 0 către localitatea *cea mai apropiată*;
- din ultima localitate vizitată se pleacă către *cea mai apropiată* localitate *nevizitată*; în caz că nu mai există localități nevizitate se revine în locul de plecare;

În abordarea euristica comis-voiajorului va alege totdeauna cea mai apropiată localitate nevizitată. Desigur, aceasta strategie nu conduce la un optim, dar considerând ca punct de plecare orice punct din rețea, pentru fiecare punct se va determina traseul minim.

```
#include<stdio.h>
void main(void)    {const   n=7;
int a[n][n], traseu[n+1], puncte[n+1], Lungime, Lungime_Min;
int i, j, punct, punct_curent, punct_min;    printf("\n Introduceți distanțele dintre puncte:\n");
for(i=0;i<n;i++)    for(j=0;j<i;j++){    printf("\n lungimea de la nodul %d la %d: ", i, j);
scanf("%d", &a[i][j]);    a[j][i]=a[i][j];    }
printf("\n Introduceți punctul de plecare:");    scanf("%d", &punct);
/* inițializarea vectorului cu punctele vizitate */
/* 0 - punct prin care nu s-a trecut */
for(i=0;i<n;i++)    puncte[i]=0;    traseu[0]=punct;    /* punctul de plecare */
puncte[punct]=1;    /* marcarea punct vizitat*/
Lungime=0; punct_curent=punct;
```



```

for(i=0;i<n-1;i++){ Lungime_min=10000;
    for(j=0;j<n;j++){ if((puncte[j]==0)&&(a[punct_curent][j]<Lungime_Min)){
        Lungime_Min=a[punct_curent][j];
        punct_min=j;}
    Lungime+=Lungime_Min;
    Traseu[i+1]=punct_min;
    Puncte[punct_min]=1;          punct_curent=punct_min;      }
    traseu[n]=punct;              Lungime+=a[punct][punct_min];      printf("\n Traseul de urmat
este:\n");
    for(i=0;i<n+1;i++)            printf("%d\t", traseu[i]);    }

```

Spre deosebire de strategiile de căutare neinformată care inspectează sistematic toate stările spațiului de căutare până în momentul găsirii stării finale, strategiile de căutare euristice încearcă reducerea numărului de stări din spațiul de căutare inspectate până la atingerea stării finale, pe baza diverselor criterii, cum ar fi funcțiile euristice. Strategia alpinistului este un exemplu de căutare informată. Alte exemple sunt strategia de căutare "best-first", algoritmul A^* și algoritmul AO^* . Algoritmii A^* și AO^* urmăresc în principal, pe lângă reducerea numărului de stări inspectate, găsirea soluției optime.

Să considerăm câteva euristici pentru problema cânilor cu apă generalizată (orice capacitate a cânilor A și B, respectiv orice rest de obținut în A):

Dacă restul este mai mare decât capacitatea câinii A, problema nu are soluție (evident)

Dacă restul nu este un multiplu al celui mai mare divizor comun al capacităților cânilor A și B, problema nu are soluție (soluția problemei rezultă din combinarea diferențelor dintre capacitățile cânilor A și B – cu o cană de 6/ și una de 3/ nu se va obține niciodată un rest de 2/, deoarece toate diferențele vor fi multipli de 3)

Deoarece rezolvarea se bazează în principal pe schimbul de apă între cele două câni pentru a găsi diferențele, există două strategii care conduc la găsirea soluției pentru orice problemă de acest tip. Notăm cu $valA$ și $valB$ cantitatea de apă la un moment dat în cana A, respectiv B și cu $capA$ și $capB$ capacitatea câinii A, respectiv B. Restul care trebuie obținut în cana A este notat cu rest

2. Analiza tehnicilor de programare

2.1 PREZENTAREA TEHNICII BACKTRACKING

2.1.1.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.

Metoda backtracking se mai poate utiliza atunci când soluția problemei se poate reprezenta sub forma unui vector $X = (x_1, x_2, \dots, x_n) \in S = S_1 \times S_2 \times \dots \times S_n$, unde S este **spațiul tuturor valorilor posibile**, iar S_i sunt mulțimi finite ordonate ($1 \leq i \leq n$). Mulțimea soluțiilor rezultat se definește astfel:

$$R = \{X \in S \mid X \text{ îndeplinește condițiile interne}\}$$

Condițiile interne sunt relații între componentele x_1, x_2, \dots, x_n ale unui vector X pe care trebuie să le îndeplinească acestea pentru ca vectorul X să fie soluție acceptabilă (corectă). Condițiile interne sunt specifice fiecărei probleme. Într-o problemă se pot cere toate soluțiile acceptabile ca rezultate sau se poate alege o soluție care maximizează sau minimizează o funcție obiectiv dată. Prin metoda backtracking nu se generează toate valorile posibile, deci nu se determină toți vectorii X din S . Se completează vectorul X secvențial; se atribuie componentei x_k o valoare numai după ce au fost atribuite deja valori componentelor x_1, x_2, \dots, x_{k-1} . Nu se va trece la componenta x_{k+1} , decât dacă sunt verificate **condițiile de continuare** pentru x_1, x_2, \dots, x_k .

Condițiile de continuare dau o condiție necesară pentru a ajunge la rezultat cu primele k alegeri făcute. Altfel spus, dacă condițiile de continuare nu sunt satisfăcute, atunci oricum am alege valori pentru următoarele componente $x_{k+1}, x_{k+2}, \dots, x_n$, nu vom ajunge la o soluție acceptabilă, deci nu are sens să continuăm completarea vectorului X . Se poate observa că cu cât aceste condiții de continuare sunt mai restrictive cu atât algoritmul va fi mai eficient, deci o bună alegere a condițiilor de continuare va duce la reducerea numărului de căutări.

Există o strânsă legătură între condițiile de continuare și condițiile interne. Condițiile de continuare se pot considera ca fiind o generalizare a condițiilor interne pentru orice subșir x_1, x_2, \dots, x_k ($1 \leq k \leq n$).

Aceasta înseamnă că pentru $k=n$ condițiile de continuare coincid cu condițiile interne. Dacă $k=n$ și sunt îndeplinite condițiile de continuare se poate da ca rezultat soluția X , pentru că au fost îndeplinite condițiile interne. Dacă nu sunt îndeplinite condițiile interne se alege altă valoare din mulțimea S_k , pentru componenta x_k , iar dacă în mulțimea S_k nu mai sunt alte valori (pentru că mulțimile sunt finite), atunci se revine la componenta anterioară x_{k-1} , încercând o nouă alegere (din mulțimea S_{k-1} pentru componenta x_{k-1}). Dacă s-a ajuns la x_1 și nu se mai poate executa o revenire ($k=0$), atunci algoritmul se termină cu insucces.

Modelul general al metodei backtracking

Rezolvarea unei probleme prin metoda backtracking se bazează pe următorul model:

Algoritmul Backtracking (X, n) este:

$k:=1$;

$x_1 :=$ “primul element din mulțimea S_1 ”;

Repetă

Cât timp ($k < n$) și “condițiile de continuare sunt îndeplinite” **execută**

$k:=k+1$;

$x_k :=$ “primul element din mulțimea S_k ”

SfCât timp

Dacă ($k=n$) și “condițiile de continuare sunt îndeplinite” **atunci**

Rezultate X

SfDacă

Cât timp ($k > 0$) și “în S_k nu mai sunt alte elemente” **execută**

$k:=k-1$

SfCât timp

Dacă $k > 0$ **atunci** $x_k :=$ “următorul element din S_k ” **SfDacă**

Până când $k=0$

SfAlgoritm.

2.2. Metoda Greedy. Algoritmii *Greedy* (greedy = lacom) sunt în general simpli și sunt folosiți în probleme de optimizare în care se poate obține optimul global în alegeri succesive ale optimului local, ceea ce permite rezolvarea problemei fără nici o revenire la deciziile anterioare. În acest sens 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; aceasta este funcția pe care dorim să o optimizăm (minim/maxim).

Modelul general de problemă tip Greedy

Se dă o mulțime A . Se cere o submulțime B ($B \subset A$) care să îndeplinească anumite condiții și să optimizeze funcția obiectiv. Dacă o submulțime C îndeplinește aceste condiții atunci C este o soluție posibilă. Este posibil ca în unele probleme să existe mai multe soluții posibile, caz în care se cere o cât mai bună soluție, eventual chiar soluția optimă, adică soluția pentru care se realizează maximul sau minimul unei funcții obiectiv.

Pentru ca o problemă să poată fi rezolvată prin metoda Greedy trebuie să îndeplinească următoarea condiție: dacă B este o soluție posibilă și $C \subset B$ atunci C este de asemenea o soluție posibilă.

Construcția soluției se realizează pas cu pas astfel:

- inițial, mulțimea B a candidaților selectați este vidă
- la fiecare pas se încearcă adăugarea celui mai promițător candidat la mulțimea B , conform funcției de selecție
- dacă după o astfel de adăugare, mulțimea B nu mai este fezabilă, se elimină ultimul candidat adăugat; acesta nu va mai fi niciodată luat în considerare
- dacă după o astfel de adăugare, mulțimea B este fezabilă, ultimul candidat adăugat va rămâne în mulțimea B
- la fiecare astfel de pas se va verifica dacă mulțimea B este o soluție posibilă a problemei.

Dacă algoritmul este corect, prima soluție găsită va fi întotdeauna o soluție optimă, dar nu neapărat unica (se poate ca funcția obiectiv să aibă aceeași valoare optimă pentru mai multe soluții posibile).

Descrierea formală a unui algoritm Greedy general este:

Algoritmul Greedy (A) este:

$B := \emptyset$;

```

Cât Timp NOT Soluție (B) AND A <>  $\phi$  Execută
    x := Select(A);
    A := A - {x};
    Dacă Fezabil(B  $\cup$  {x}) Atunci B := B  $\cup$  {x}
    SfDacă
SfCât
Dacă Soluție(B) Atunci "B este soluție"
    Altfel "Nu există soluție"
SfDacă

```

SfAlgoritm

Soluție(B) - funcție care verifică dacă o mulțime de candidați e o soluție posibilă;

Select(A) - funcție derivată din funcția obiectiv

Fezabil(B) - funcție care verifică dacă o mulțime de candidați este fezabilă.

Această metodă este utilă în rezolvarea problemelor în care se cere să se aleagă o mulțime de elemente care să satisfacă anumite condiții și să realizeze un optim: să se găsească cea mai bună ordine de efectuare a unor lucrări într-un timp minim, să se găsească cel mai scurt drum într-un graf, problema restului cu număr minim de monezi, interclasarea optimă a șirurilor ordonate, problema comis-voiajorului, problema rucsacului.

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 ← 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 exista 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.

Observații

1. este necesar ca înaintea aplicării unui algoritm GREEDY să se demonstreze că acesta furnizează soluția optimă
2. tehnica GREEDY conduce la un timp de calcul polinomial. În marea majoritate, algoritmi GREEDY au complexitate $O(n^2)$
3. sunt multe aplicații care se rezolvă folosind această metodă: determinarea arborelui parțial minim într-un graf (algoritmul lui Kruskal), determinarea lanțului de lungime minimă între două vârfuri ale unui graf neorientat (algoritmul lui Dijkstra), determinarea drumului de lungime minimă între două vârfuri ale unui graf orientat (algoritmul lui Bellman-Kalaba)
4. nu întotdeauna există algoritmi GREEDY (sau algoritmi polinomiali) pentru găsirea soluției optime a unei probleme. În acest caz se caută soluții apropiate de soluția optimă, obținute în timp util. Astfel de algoritmi se numesc algoritmi euristici

2.3 METODA GREEDY COMBINATĂ CU BACKTRACKING

Exemplu 5. Problema rucsacului. Se dau n obiecte care se pun la dispoziția unei persoane care le poate transporta cu ajutorul unui rucsac. Greutatea maximă admisă este G . Pentru fiecare obiect i se notează prin $c_i > 0$ câștigul obținut în urma transportului său în întregime la destinație, iar prin $g_i > 0$ greutatea obiectului i .

Să se facă o încărcare a rucsacului astfel încât câștigul să fie maxim știind că sau:

1) pentru orice obiect putem încărca orice parte din el în rucsac; sau

2) orice obiect poate fi încărcat numai în întregime în rucsac.

R. 1) *Problema continuă*. Pentru a obține soluția optimă în acest caz e suficient să aplicăm metoda Greedy, încărcând în rucsac obiectele în ordinea descrescătoare a valorilor c_i/g_i (reprezentând câștigurile pe unitatea de greutate), până când acesta se umple. Pentru a ști ordinea folosim o procedură care construiește vectorul k : al pozițiilor în șirul ordonat descrescător. De exemplu: dacă $c=(50, 10, 20)$ și $g=(25, 1, 5)$ atunci $c/g=(2, 10, 4)$ și deci $k^{-1}=(3, 1, 2) \Rightarrow k=(2, 3, 1)$.

Algoritmul este dat în procedura *RucsacCont*.

Procedure RucsacCont($g, c, G, n; x$)

array $g(n), c(n)$

call Numărare($c, g, n; k$); $G1:=G$

for $i=1, n$:

if $G1 > g(k_i)$ **then** $x(k_i) := 1$

else $x(k_i) := G1/g(k_i)$

endif

$G1 := G1 - x(k_i)g(k_i)$

repeat

return; end

Procedure Numărare($c, g, n; k$)

array $c(n), g(n), k(n), kinv(n)$

for $i=1, n$:

$kinv(i) := 1$

repeat

for $i=1, n$:

for $j=i+1, n$:

if $c/g_i < c/g_j$ **then** $kinv(i) := kinv(i) + 1$

else $kinv(j) := kinv(j) + 1$

endif

repeat

repeat

for $i=1, n$:

$k(kinv(i)) := i$

repeat

return

end

Încercați să eliminați vectorul $kinv$ și să folosiți pentru inversare doar o variabilă.

2.4. Metoda Divide et Impera. *Divide et Impera* este o metodă ce constă în:

- descompunerea problemei ce trebuie rezolvată într-un număr de subprobleme mai mici ale aceleiași probleme

- rezolvarea succesivă și independentă a fiecăreia dintre subprobleme

- recompunerea subsoluțiilor astfel obținute pentru a găsi soluția problemei inițiale.

Modelul general al metodei Divide et Impera

Presupunând ca se dorește determinarea soluției x a unei probleme ale cărei date sunt păstrate într-un tablou între indicii p și q , procedura DIVIDE ET IMPERA poate fi descrisă în felul următor:

procedura D&I (p, q, x)

{ *daca dimensiunea problemei este mare*

atunci

{ *imparte*(p, q, r); // se descompune problema

D&I($p, r, x1$); // se rezolva prima subproblema

D&Ie($r+1, q, x2$); // se rezolva a doua subproblema

combinare($x1, x, x$) // se combina subsolutiile }

altfel *soluție*(p, q, x) // problema admite rezolvare imediata }

Observație

- problema poate fi descompusa în mai multe subprobleme (nu neapărat în doua) și apoi subsoluciile acestora se vor combina

Multe probleme se rezolva folosind tehnica *DIVIDE ET IMPERA*: căutarea binară a unei valori într-un sir, parcurgerile arborilor binari, sortarea prin interclasare a unui sir de numere, sortarea rapida a unui sir de numere.

Algoritmul Divide_et_Impera(x) este:

```
{ returnează o soluție pentru cazul x }
Dacă "x este suficient de mic" atunci adhoc(x) SfDacă
{ descompune x în subcazurile  $x_1, x_2, \dots, x_k$  }
Pentru  $i:=1, k$  execută  $y_i := divimp(x_i)$  SfPentru
{ recompune  $y_1, y_2, \dots, y_k$  în scopul obținerii soluției y pentru x }
```

Rezultate: y

unde: *adhoc* - este subalgoritmul de bază pentru rezolvarea subcazurilor mici ale problemei în cauză; *divimp* - împarte probleme în subprobleme.

Un algoritm divide et impera trebuie să evite descompunerea recursivă a subcazurilor "suficient de mici", deoarece pentru acestea este mai eficientă aplicarea directă a algoritmului de bază. Când se poate însă decide că un caz este "suficient de mic"?

În general, se recomandă o metodă hibridă care constă în:

- determinarea teoretică a formulei recurente
- găsirea empirică a valorilor constante folosite de aceste formule în funcție de implementare.

Se observă că această metodă este prin definiție recursivă. Uneori este posibilă eliminarea recursivității printr-un ciclu iterativ.

2.5 Euristica greedy. (Vezi m. sos 4.3 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 consta în descompunerea procesului de căutare a soluției optime în mai multe subproces succesive, fiecare din aceste subproces 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 implica, în general, optimizarea globală. Regăsim, de fapt, principiul care sta la baza metodei greedy. Un algoritm greedy, despre care nu se poate demonstra că furnizează soluția optimă, este un algoritm euristic.

Vom da doua exemple de utilizare a algoritmilor greedy euristici.

2.5.1 Colorarea unui graf. Fie $G = \langle V, M \rangle$ un graf neorientat, ale cărui vârfuri trebuie colorate astfel încât oricare doua 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 ramase, încercând să le colorăm, fără a schimba culoarea. Când nici un vârf nu mai poate fi colorat, schimbăm culoarea și vârfurile de start, repetând procedeul.

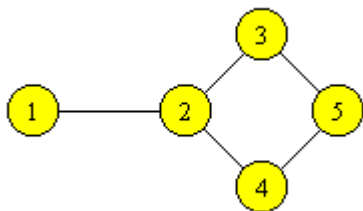


Figura 1 Un graf care va fi colorat.

Dacă în graful din Figura 1 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 euristica, 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 harta oarecare trebuie colorata cu un număr minim de culori, astfel încât doua tari cu frontiera comuna sa fie colorate diferit. Daca fiecărui vârf îi corespunde o tara, iar doua vârfuri adiacente reprezintă tari cu frontiera comuna, atunci hărții îi corespunde un graf *planar*, adică un graf care poate fi desenat în plan fără ca doua muchii sa se intersecteze. Celebritatea problemei consta în faptul ca, în toate exemplele întâlnite, colorarea s-a putut face cu cel mult 4 culori. Aceasta în timp ce, teoretic, se putea demonstra ca pentru o harta oarecare este nevoie de cel mult 5 culori. Recent [4] s-a demonstrat pe calculator faptul ca orice harta poate fi colorata cu cel mult 4 culori. Este prima demonstrare pe calculator a unei teoreme importante.

Problema colorării unui graf poate fi interpretata și în contextul planificării unor activități. De exemplu, sa presupunem ca dorim sa executam simultan o mulțime de activități, în cadrul unor săli de clasa. 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.

6.10.2 Problema comis-voiajorului. Se cunosc distantele dintre mai multe orașe. Un comis-voiajor pleacă dintr-un oraș și dorește sa se întoarcă în același oraș, după ce a vizitat fiecare din celelalte orașe exact o data. Problema este de a minimiza lungimea drumului parcurs. și pentru aceasta problema, toți algoritmi care găsesc soluția optima sunt exponențiali.

Problema poate fi reprezentata printr-un graf neorientat, în care oricare doua vârfuri diferite ale grafului sunt unite între ele printr-o muchie, de lungime nenegativa. Căutăm un ciclu de lungime minima, care sa se închidă în vârful inițial și care sa treacă prin toate vârfurile grafului.

Conform strategiei greedy, vom construi ciclul pas cu pas, adăugând la fiecare iterație cea mai scurta muchie disponibila cu următoarele proprietăți:

- nu formează un ciclu cu muchiile deja selectate (exceptând pentru ultima muchie aleasa, care completează ciclul)

- nu exista încă doua muchii deja selectate, astfel încât cele trei muchii sa fie incidente în același vârf

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 1. Matricea distantelor pentru problema comis-voiajorului.

De exemplu, pentru șase orașe a căror matrice a distantelor este data în Tabelul 1, 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.

După cum spuneam, exista probleme pentru care aplicarea strategiei GREEDY nu conduce întotdeauna la soluția optima. Un astfel de algoritm se numește **algoritm euristic**.

În marea majoritate, problemele la care se încearcă găsirea unor algoritmi euristici sunt problemele pentru a căror rezolvare nu se cunosc algoritmi polinomiali. În acest caz se prefera un algoritm euristic, care găsește soluții “apropiate” de soluția optima, unui algoritm exponențial, al cărui timp de execuție este foarte mare.

2.6. Metoda programării dinamice *Programarea dinamica*, ca și metoda divide et impera, rezolva problemele combinând soluțiile subproblemelor. După cum am văzut, algoritmi divide et impera partiționează problemele în subprobleme independente, rezolva subproblemele în mod recursiv, iar apoi combina soluțiile lor pentru a rezolva problema inițială. Daca subproblemele conțin subsubprobleme comune, în locul metodei divide et impera este mai avantajos de aplicat tehnica programării dinamice.

Sa analizam însă pentru început ce se întâmplă cu un algoritm divide et impera în aceasta din urma situație. Descompunerea recursiva 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 eficienta scăzută a algoritmului. Sa ne amintim, de exemplu, de algoritmul *fib1* din Capitolul 1. Sau, sa calculam 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, rezulta ca 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 | | | |
| \vdots | | | | | | |
| $n-1$ | | | | | $\binom{n-1}{k-1}$ | $\binom{n-1}{k}$ |
| 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 *fib2*. 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, până 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 intra 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.

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:

- $\square\square\square$ se caracterizează structura unei soluții optime
- $\square\square\square$ se definește recursiv valoarea unei soluții optime
- $\square\square\square$ 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:
- $\square\square\square$ 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.

Cu alte cuvinte această metodă de elaborare a algoritmilor se aplică în rezolvarea problemelor de optim pentru care soluția este rezultatul unui șir de decizii. Ca și metoda divide et impera, programarea dinamică rezolvă problemele combinând soluțiile subproblemelor. Dacă subproblemele conțin sub-subprobleme comune atunci este mai eficientă aplicarea tehnicii programării dinamice.

Metoda programării dinamice se caracterizează prin trei principii fundamentale:

- evitarea calculării de mai multe ori a aceluiași subcaz, prin memorarea rezultatelor intermediare;
- programarea dinamică construiește algoritmul de jos în sus: se pornește de la cele mai mici subcazuri. Combinând soluțiile lor, se obțin soluții pentru subcazuri din ce în ce mai mari, până se ajunge la soluția cazului inițial;

•principiul optimalității: într-o secvență optimă de decizii, fiecare subsecvență trebuie să fie de asemenea optimă.

Ca și în cazul algoritmilor greedy, soluția optimă nu este neapărat unică.

Modelul general al metodei programării dinamice

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.

Problema înmulțirii înlanțuite optime a matricelor se poate rezolva și prin următorul algoritm recursiv:

function *rminscal*(*i, j*)

{returnează numărul minim de înmulțiri scalare
pentru a calcula produsul matricial $M_i M_{i+1} \dots M_j$ }

if $i = j$ **then return** 0

$q \leftarrow +\infty$

for $k \leftarrow i$ **to** $j-1$ **do**

$q \leftarrow \min(q, \text{rminscal}(i, k) + \text{rminscal}(k+1, j) + d[i-1]d[k]d[j])$

return q

unde tabloul $d[0 \dots n]$ este global. Găsiți o limita 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 inducției, deduceți ca $r(n) \geq 2^{n-1}$, pentru $n > 2$. Timpul pentru un apel *rminscal*(1, *n*) este atunci în $\Omega(2^n)$.

2.6.1 Determinarea celor mai scurte drumuri într-un graf. Fie $G = \langle V, M \rangle$ 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.

Vom presupune că vârfurile sunt numerotate de la 1 la n și că matricea L da lungimea fiecărei muchii: $L[i, i] = 0$, $L[i, j] \neq 0$ 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 $\{1, 2, \dots, k\}$. 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 $\{1, 2, \dots, k-1\}$. 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*($L[1 \dots n, 1 \dots n]$)

array $D[1 \dots n, 1 \dots n]$

$D \leftarrow L$

for $k \leftarrow 1$ **to** n **do**


```

for  $i \leftarrow 1$  to  $n$  do
for  $j \leftarrow 1$  to  $n$  do
 $D[i, j] \leftarrow \min(D[i, j], D[i, k] \square D[k, j])$ 
return  $D$ 

```

De exemplu, daca avem :

$$D=L= \begin{array}{ccccc} 0 & 2 & _ & 10 & _ \\ 2 & 0 & 3 & _ & _ \\ _ & 3 & 0 & 1 & 8 \\ 10 & _ & 1 & 0 & _ \\ _ & _ & 8 & _ & 0 \end{array}$$

obținem succesiv :

$$D1= \begin{array}{ccccc} 0 & 2 & \mathbf{5} & 10 & _ \\ 2 & 0 & 3 & 12 & _ \\ \mathbf{5} & 3 & 0 & 1 & 8 \\ 10 & 12 & 1 & 0 & _ \\ _ & _ & 8 & _ & 0 \end{array} \quad D2= \begin{array}{ccccc} 0 & 2 & 5 & 6 & \mathbf{13} \\ 2 & 0 & 3 & 12 & _ \\ 5 & 3 & 0 & 1 & 8 \\ 6 & 12 & 1 & 0 & _ \\ \mathbf{13} & _ & 8 & _ & 0 \end{array}$$

$$D3= \begin{array}{ccccc} 0 & 2 & 5 & 6 & 13 \\ 2 & 0 & 3 & \mathbf{4} & _ \\ 5 & 3 & 0 & 1 & 8 \\ 6 & \mathbf{4} & 1 & 0 & _ \\ 13 & _ & 8 & _ & 0 \end{array} \quad D4= \begin{array}{ccccc} 0 & 2 & 5 & 6 & 13 \\ 2 & 0 & 3 & 4 & 11 \\ 5 & 3 & 0 & 1 & 8 \\ 6 & 4 & 1 & 0 & \mathbf{9} \\ 13 & 11 & 8 & \mathbf{9} & 0 \end{array}$$

Puteți deduce ca algoritmul lui Floyd necesita un timp în $\square(n^3)$. Un alt mod de a rezolva aceasta problema este sa aplicam algoritmul *Dijkstra* (Capitolul 6) de n ori, alegând mereu un alt vârf sursa. Se obține un timp în $n \square(n^2)$, adică tot în $\square(n^3)$. Algoritmul lui Floyd, datorita simplității lui, are însă constanta multiplicativa mai mica, fiind probabil mai rapid în practica.

Daca folosim algoritmul *Dijkstra-modificat* în mod similar, obținem un timp total în $O(\max(mn, n^2) \log n)$, unde $m = \#M$. Daca graful este rar, atunci este preferabil sa aplicam algoritmul *Dijkstra-modificat* de n ori; daca graful este dens ($m \square n^2$), este mai bine sa folosim algoritmul lui Floyd.

De obicei, dorim sa aflam nu numai lungimea celui mai scurt drum, dar și traseul sau. în acesta situație, vom construi o a doua matrice P , inițializată cu zero. Bucla cea mai interioara a algoritmului devine

```

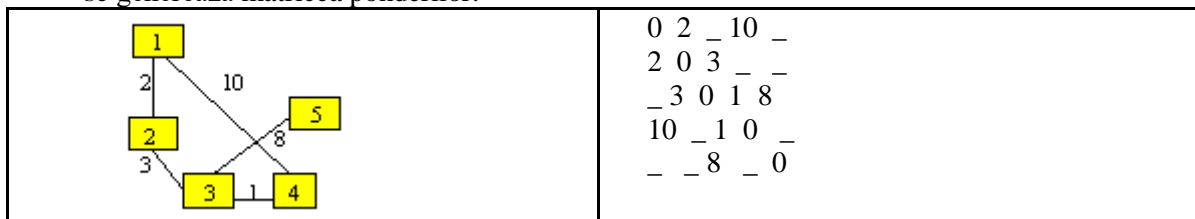
if  $D[i, k] \square D[k, j] < D[i, j]$  then  $D[i, j] \leftarrow D[i, k] \square D[k, j]$ 
 $P[i, j] \leftarrow k$ 

```

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 , consultam elementul $P[i, j]$. Daca $P[i, j] = 0$, atunci cel mai scurt drum este chiar muchia (i, j) . Daca $P[i, j] = k$, atunci cel mai scurt drum de la i la j trece prin k și urmează sa consultam recursiv elementele $P[i, k]$ și $P[k, j]$ pentru a găsi și celelalte vârfuri intermediare.

Algoritmul:

-se generează matricea ponderilor:



Pentru exemplul precedent se obține :

$$\begin{array}{ccccc} 0 & 2 & 5 & 6 & 13 \\ 2 & 0 & 3 & 4 & 11 \end{array}$$

```

5 3 0 1 8
6 4 1 0 9
13 11 8 9 0

```

Astfel matricea costurilor va arata ca în partea dreapta. Afișarea drumului de la un nod la altul se face cu ajutorul funcției `descompun_drum` ce se bazează pe metoda `divide_et_impera`. De exemplu pentru un drum de la nodul 1 la nodul 4 avem inițial calea directă 1-4 cu costul 10 dar tot odată mai există o cale prin nodurile intermediare cu costul mai mic de valoare 6.

2.7. Metoda Branch and Bound folosește la rezolvarea problemelor la care domeniul în care se caută soluția este foarte mare și nu se cunoaște un alt algoritm care să conducă mai rapid la rezultat. Problemele care pot fi abordate prin această metodă pot fi modelate într-un mod asemănător celui folosit la metoda Backtracking. Se pleacă de la o configurație inițială și se reține șirul de operații prin care aceasta este transformată într-o configurație finală dacă aceasta este dată, în alte cazuri se cere configurația finală știind că trebuie să verifice anumite condiții de optim.

Diferența dintre cele două metode constă în faptul că metoda Backtracking la fiecare etapă selectează un singur succesori după care se face verificarea condițiilor de continuare, iar metoda Branch and Bound generează la fiecare pas toate configurațiile posibile (toți succesorii) care rezultă din cea curentă și le stochează într-o listă. Se alege apoi un element din listă după un criteriu ce depinde de problemă și se reia procesul de expansiune.

Alegerea optimă a unui element din această listă pentru expansiune se face cu ajutorul unei funcții $f = g + h$ în care g este o funcție care măsoară lungimea drumului parcurs de la configurația inițială până la nodul curent iar h este o funcție (euristică) care estimează efortul necesar până se ajunge la soluție și este specifică fiecărei probleme. Alegerea funcției h este foarte importantă din punct de vedere a vitezei de execuție a programului.

Se lucrează cu două liste: lista open în care se rețin configurațiile neexpandate încă și lista close care le memorează pe cele expandate. Soluția se construiește folosind un arbore care se parcurge pe baza legăturii tată.

Nodurile sunt înregistrări care cuprind următoarele informații:

- configurația la care s-a ajuns, t ;
- valorile funcțiilor g și h ,
- adresa nodului 'tată';
- adresa înregistrării următoare în lista open sau close;
- adresa înregistrării următoare în lista succesorilor.

Algoritmul:

1. înregistrarea corespunzătoare configurației inițiale este încărcată în open cu $g=0$ și $f=h$;
2. atât timp cât nu s-a selectat spre expansiune nodul corespunzător configurației finale și lista open este nevidă, se execută următoarele:
3. se selectează din lista open nodul t cu f minim;
4. se expandează acest nod obținând o listă liniară simplă înălțată cu succesorii săi;
5. pentru fiecare succesori din această listă se execută:
6. se atașează noul g , obținut ca sumă între valoarea lui g a configurației t și costul expansiunii (de obicei 1);
7. se testează dacă acest succesori aparține listei open sau close și în caz afirmativ, se verifică dacă valoarea lui g este mai mică decât cea a configurației găsite în listă;
8. în caz afirmativ, nodul găsit este direcționat către actualul părinte (prin fixarea legăturii tată) și se atașează noul g , iar dacă acest nod se găsește în close, este trecut în open;
9. în caz că acest nod nu se găsește în open sau close, este introdus în lista open;
10. dacă s-a selectat pentru expansiune nodul corespunzător configurației finale atunci se trasează folosind o procedură recursivă drumul de la configurația inițială la cea finală utilizând legătura 'tată';
11. dacă ciclul se încheie deoarece lista open este vidă înseamnă că problema nu are soluție.

Obs: Algoritmul se poate implementa folosind o singură listă iar diferențierea dintre nodurile expandate și cele neexpandate se va face după un câmp special. Apare însă dezavantajul că operațiile de căutare se fac într-o listă mai lungă. Acest dezavantaj se poate elimina folosind o structură specială numită tabelă hash. În unele cazuri se poate folosi un arbore binar.

PREZENTAREA TEHNICII. "Branch and Bound" (BB) este o tehnică de programare ce se aplică problemelor a căror rezolvare necesită un timp mare de execuție, și în plus nu se cunosc algoritmi de rezolvare care să furnizeze mai rapid soluția.

3. Probleme cu exemple pentru însușire, modificare și rularea aplicațiilor:

Probl.1. Considerăm următoarea problemă (Săritura calului):

Se dă o tablă de șah. Se cere să se găsească un drum pe care îl parcurge un cal care pornește dintr-un colț și sare într-o poziție liberă (în care nu a mai sărit) conform mutării de la jocul de șah. În acest mod, calul va trebui să treacă prin toate căsuțele. În cazul în care problema nu are soluție se va afișa un mesaj corespunzător.

Rezolvare: În primul rând, se ține de cont de modul în care se poate deplasa un cal pe tabla de șah. Față de o anumită poziție, mutările calului implică modificarea coordonatelor pe tabla de șah conform matricei **Mutari**. Apoi, se pornește din colțul de coordonate (1,1) și se va încerca înaintarea. Condițiile de continuare conțin: calul să nu depășească marginile tablei și poziția pe care o va ocupa în urma mutării să nu fie deja ocupată. Funcția **Rezolva** va specifica dacă problema are soluție.

```
program SarituraCalului;
const MaxDim = 8; { Dimensiunea maxima a tablei de șah }
Mutari: array[1 .. 8, 1 .. 2] of Integer = ((-1, -2), (-1, 2), (1, -2), (1, 2), (-2, -1), (-2, 1), (2, -1), (2, 1));
var Tabla: array[1 .. MaxDim, 1 .. MaxDim] of Integer; { Tabla de șah }
Dim: Integer; { Dimensiunea tablei }
NrMutari: Integer; { Numărul mutărilor }
NrSolutii: Integer; { Numărul soluțiilor }
{ Afișarea unei soluții găsite } procedure Afis;
var I, J: Integer;
begin
  for I := 1 to Dim do
    begin
      for J := 1 to Dim do
        Write(Tabla[I, J] : 5); WriteLn; end; WriteLn;
      end;
    { Procedura recursiva } procedure Recurs(X, Y: Integer);
var DX, DY: Integer; { Coordonatele rezultate în urma deplasării }
I: Integer;
begin
  NrMutari := NrMutari + 1; Tabla[X, Y] := NrMutari;
  if NrMutari = Dim * Dim then
    begin
      NrSolutii := NrSolutii + 1; Afis; end;
    for I := 1 to 8 do
      begin
        DX := X + Mutari[I][1]; DY := Y + Mutari[I][2];
        if (DX >= 1) and (DX <= Dim) and (DY >= 1) and (DY <= Dim) and
          (Tabla[DX, DY] = 0) then { Daca mutarea nu depășește marginile tablei,
            și careul nu a mai fost ocupat înainte } Recurs(DX, DY);
        end;
        { Se retrage mutarea, eliberând careul (X, Y) }
        NrMutari := NrMutari - 1; Tabla[X, Y] := 0;
      end;
    { Rezolvarea problemei } function Rezolva: Boolean;
var I, J: Integer;
begin
  if Dim < 3 then { Nu se poate efectua nici o mutare } Rezolva := False
  else begin { Inițializări }
    if Dim > MaxDim then Dim := MaxDim;
    for I := 1 to Dim do
      for J := 1 to Dim do
        Tabla[I, J] := 0; NrMutari := 0; NrSolutii := 0;
        { Apelul metodei backtracking } Recurs(1, 1); { Se pornește din careul stânga-sus }
        Rezolva := (NrSolutii > 0); end;
      end;
    { Programul principal } begin
      repeat Write('Dimensiunile tablei: '); ReadLn(Dim); until (Dim > 0) and (Dim <= MaxDim);
      if Rezolva then WriteLn(NrSolutii, ' soluții.') else WriteLn('Problema nu are soluții.'). end.
```

Probl.2. Pentru exemplificarea mai largă, vom considera o problema 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 metoda 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 , $1 \leq i \leq 8$, se afla 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 coloana și, de asemenea, exista doua perechi de regine situate pe aceeași diagonală. Folosind acesta reprezentare, putem scrie în mod direct algoritmul care găsește o soluție a problemei:

```

procedure regine1
  for  $i_1 \leftarrow 1$  to 8 do
    for  $i_2 \leftarrow 1$  to 8 do
       $\vdots$ 
      for  $i_8 \leftarrow 1$  to 8 do           $posibil \leftarrow (i_1, i_2, \dots, i_8)$ 
        if  $soluție(posibil)$  then write  $posibil$     stop
  write “nu exista soluție”

```

De aceasta data, 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 noua îmbunătățire. Dacă introducem și restricția ca doua regine sa nu se afle pe aceeași coloana, o configurație pe tabla de șah se poate reprezenta ca o permutare a primilor opt întregi. Algoritmul devine

```

procedure regine2
   $posibil \leftarrow$  permutarea inițială
  while  $posibil \neq$  permutarea finală and not  $soluție(posibil)$  do
     $posibil \leftarrow$  următoarea permutare
  if  $soluție(posibil)$  then write  $posibil$ 
    else write “nu exista 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 data recursiv toate permutările celor $n-1$ elemente ramase:

```

procedure perm( $i$ )
  if  $i = n$  then  $utilizeaza(T)$  { $T$  este o noua permutare}
    else for  $j \leftarrow i$  to  $n$  do  $interschimba\ T[i]\ \text{și}\ T[j]$ 
       $perm(i+1)$ 
       $interschimba\ T[i]\ \text{și}\ T[j]$ 

```

În algoritmul de generare a permutărilor, $T[1..n]$ este un tablou global inițializat cu $[1, 2, \dots, n]$, iar primul apel al procedurii este $perm(1)$. Dacă $utilizeaza(T)$ necesita un timp constant, atunci $perm(1)$ necesita un timp în $\Theta(n!)$.

Aceasta abordare reduce numărul de configurații posibile la $8! = 40.320$. Dacă se folosește algoritmul $perm$, atunci pana 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 exista doua 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 tabla. 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 problema de căutare într-un arbore. Spunem că vectorul $P[1..k]$ de întregi între 1 și 8 este k -promițător, pentru $0 \leq k \leq 8$, dacă zonele controlate de cele k regine plasate în pozițiile $(1, P[1]), (2, P[2]), \dots, (k, P[k])$ sunt disjuncte. Matematic, un vector P este k -promițător dacă:

$$P[i] - P[j] \notin \{i - j, 0, j - i\}, \quad \text{pentru orice } 0 \leq i, j \leq k, i \neq j$$

Pentru $k \leq 1$, orice vector P este k -promițător. Soluțiile problemei celor opt regine corespund vectorilor 8-promitatori.

Fie V mulțimea vectorilor k -promițători, $0 \leq k \leq 8$. Definim graful orientat $G = \langle V, M \rangle$ astfel: $(P, Q) \in M$, dacă și numai dacă exista un întreg k , $0 \leq k \leq 8$, astfel încât P este k -promițător, Q este $(k+1)$ -promițător și $P[i] = Q[i]$ pentru fiecare $0 \leq i \leq k$. 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 regina 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

parcure 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 *regine2*. 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 *regine2*, 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  $j \leftarrow 1$  to 8 do
        if plasare( $k, j$ ) then posibil[ $k+1$ ]  $\leftarrow j$ 
          regine( $k+1$ )

  function plasare( $k, j$ )
    {returnează true, dacă și numai dacă se
     poate plasa o regină în poziția  $(k+1, j)$ }
    for  $i \leftarrow 1$  to  $k$  do
      if  $j \neq \text{posibil}[i] \wedge \{k+1-i, 0, i-k+1\}$  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 *regine2* 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 întâlnite 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 legătura directă între ele și nu se cere ca ciclul să fie optim.

Parcurerea 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 manipularilor 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 manipulari.

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.//////////



Aceasta tehnica de programare se folosește în rezolvarea problemelor care satisfac următoarele condiții:

- soluția problemei se reprezintă sub forma unui vector $x=(x_1, \dots, x_n)$ unde $x_1 \in A_1, \dots, x_n \in A_n$
- mulțimile A_1, \dots, A_n sunt mulțimi finite și ordonate
- elementele vectorului soluție x trebuie să satisfacă un set de condiții, numite **condiții interne**
- nu se dispune de o alta metoda de rezolvare, mai rapida.

Observații

- de multe ori nu se cunoaște numărul n al elementelor din soluție, ci o condiție ce trebuie să o satisfacă elementele vectorului x , numita **condiție finală**.
- componentele vectorului x , x_1, \dots, x_n pot fi la rândul lor vectori
- în cele mai multe situații, mulțimile A_1, \dots, A_n coincid
- problemele care se pot rezolva utilizând aceasta tehnica sunt cele așa numite **nedeterminist polinomiale**, pentru care nu se cunoaște o soluție iterativă.

Principiul de generare a soluției

- soluția se construiește pas cu pas: x_1, x_2, \dots, x_n
- presupunând generate la un moment dat elementele x_1, x_2, \dots, x_k , aparținând mulțimilor A_1, \dots, A_k se alege (daca exista) x_{k+1} , primul element neales încă din mulțimea A_{k+1} și care verifică condițiile de continuare, altfel spus subsoluția x_1, x_2, \dots, x_{k+1} să satisfacă **condițiile interne**. Apar doua posibilități:
 -  nu s-a găsit un astfel de element, se revine, considerând generata subsoluția x_1, \dots, x_{k-1} și se caută o alta valoare pentru x_k - următorul element netestat încă din mulțimea A_k .
 -  s-a găsit o valoare corespunzătoare pentru x_{k+1} , caz în care apar din nou doua situații:
 - i. s-a ajuns la soluție, se tipărește soluția și se reia algoritmul considerând generata subsoluția x_1, \dots, x_k (se caută pentru x_{k+1} un element din mulțimea A_{k+1} rămas netestat)
 - ii. nu s-a ajuns încă la soluție, caz în care se considera generata subsoluția x_1, \dots, x_{k+1} și se caută un prim element $x_{k+2} \in A_{k+2}$.

Algoritmul se termina atunci când au fost testate toate elementele din A_1 .

Observații

- în urma aplicării tehnicii backtracking, se generează toate soluțiile problemei. În cazul în care se cere doar o singura soluție, se poate opri generarea atunci când a fost găsită
- alegerea condițiilor de continuare este foarte importanta, deoarece duce la micșorarea numărului de calcule
- problemele rezolvate folosind tehnica backtracking necesita un timp de execuție îndelungat. Spre exemplu, presupunând ca $A_1 = A_2 = \dots = A_n = \{1, 2, \dots, n\}$, numărul total al subsoluțiilor care se pot obține - fără a lua în considerare condițiile interne - este n^n . Chiar dacă nu se poate evalua exact - depinde de condițiile interne alese -, timpul de execuție crește exponențial, pe măsura creșterii lui n .

În continuare vom prezenta variante ale tehnicii backtracking și o serie de probleme rezolvate prin aceasta metoda.

Aceasta metodă generală de programare se aplică problemelor în care soluția se poate reprezenta sub forma unui vector $\chi = (\chi_1, \dots, \chi_n) \in S = S_1 \times \dots \times S_n$ unde mulțimile S_1, \dots, S_n sunt mulțimi finite având $|S_i| = s_i$ și elemente. Pentru fiecare problema concretă sunt date anumite relații între componentele χ_1, \dots, χ_n ale vectorului χ , numite **condiții interne**.

Mulțimea finită $S = S_1 \times \dots \times S_n$ se numește *spațiul soluțiilor posibile*. Soluțiile posibile care satisfac condițiile interne se numesc *soluții rezultat*. Ceea ce ne propunem este de a determina toate soluțiile rezultat, cu scopul de a le lista sau de a alege dintre ele una care maximizează sau minimizează o eventuală funcție obiectiv dată.

O metodă simplă de determinare a soluțiilor rezultat constă în a genera într-un mod oarecare toate soluțiile posibile și de a verifica dacă ele satisfac condițiile interne. Dezavantajul constă în faptul că timpul cerut de aceasta investigare exhaustivă este foarte mare. Astfel chiar pentru $|S_i| = 2, \forall i$, timpul necesar este $\theta(2^n)$, deci exponențial. Deci având un timp de execuție exponențial, utilizarea metodei backtracking se justifică numai atunci când nu cunoaștem o altă metodă cu eficiența mai mare sau avem de rezolvat probleme în care se cere generarea tuturor soluțiilor posibile.

Metoda backtracking urmărește să evite generarea tuturor soluțiilor posibile. În acest scop elementele vectorului χ primesc pe rând valori, în sensul că lui χ_k i se atribuie o valoare numai dacă au fost atribuite deja valori lui $\chi_1, \dots, \chi_{k-1}$. Mai mult, odată o valoare pentru χ_k stabilită, nu se trece direct la

atribuirea de valori lui χ_{k+1} , ci se verifică niște *condiții de continuare* referitoare la χ_1, \dots, χ_k . Aceste condiții stabilesc situațiile în care are sens să trecem la calculul lui χ_{k+1} , neîndeplinirea lor exprimând faptul că oricum am alege $\chi_{k+1}, \dots, \chi_n$ nu vom putea ajunge la o soluție rezultat, adică la o soluție pentru care condițiile interne să fie satisfăcute. Este evident că în cazul neîndeplinirii condițiilor de continuare va trebui să facem o altă alegere pentru χ_k sau, dacă S_k a fost epuizat, să micșorăm pe k cu o unitate încercând să facem o nouă alegere pentru χ_k etc.; aceasta micșorare a lui k dă numele metodei, ilustrând faptul că atunci când nu putem avansa, urmărim înapoi secvența curentă din soluție. Este evident că între condițiile de continuare și condițiile interne există o strânsă legătură. O bună alegere pentru condițiile de continuare are ca efect o importantă reducere a numărului de calcule.

Probl.3. Presupunând condițiile de continuare stabilite, putem scrie procedura BACKTRACK, în care $\varphi_k(\chi_1, \dots, \chi_k)$ reprezintă condițiile de continuare pentru χ_1, \dots, χ_k .

```

procedure BACKTRACK (n, x)
array x(n)
k ← 1
while k > 0:
    i ← 0
    while [mai există o valoare α din Sk netestată]
        χk ← α
        if φk(x1, ..., xk) then i ← 1; exit
        endif
    repeat
    if i = 0 then k ← k - 1
        else if k = n then write x1, ..., xn
            else k ← k + 1
        endif
    endif
repeat
return
end

```

Un caz particular important este cel în care pentru fiecare k , S_k este o progresie aritmetică $S_k = \{a_k, a_k + h_k, a_k + 2h_k, \dots, b_k\}$. Atunci presupunând $h_k > 0, \forall k$, se obține procedura BACKTRACK1.

```

procedure BACKTRACK1 (n, a, b, h)
real a(n), b(n), h(n), x(n)
k ← 1; xk ← a1 - h
while k > 0 i ← 0
    while xk + hk ≤ bk:
        xk ← xk + hk
        if φk(x1, ..., xk) then i ← 1; exit
        endif
    repeat
    if i = 0 then k ← k - 1 else if k = n then write x1, ..., xn
        else k ← k + 1; xk ← ak - hk
    endif
    endif
repeat
return
end

```

Să observăm că dacă în algoritmiile de mai sus înlocuim predicatul $\varphi_k(\chi_1, \dots, \chi_k)$ cu 1, atunci vor fi listați toți vectorii din $S = S_1 \times \dots \times S_n$.

Metoda backtracking poate fi reprezentată ușor pe un arbore construit astfel:

- nivelul 1 conține rădăcina;
 - din orice vârf de pe nivelul k pleacă s_k muchii spre nivelul $k + 1$, etichetate cu cele s_k elemente ale lui S_k .
- Nivelul $n + 1$ va conține $s_1 \cdot s_2 \cdot \dots \cdot s_n$ vârfuri. Pentru fiecare vârf de pe nivelul $n + 1$, etichetele muchiilor conținute în drumul ce leagă rădăcina de acest vârf reprezintă o soluție posibilă.

Să observăm că modul de construcție a soluțiilor posibile folosit de metoda backtracking corespunde parcurgerii DF (down first) a arborilor.

Probl.4. Exemplu. Să considerăm *problema submulțimilor de suma dată* care constă în următoarele: Fie $A = (a_1, \dots, a_n)$ cu $a_i > 0, \forall i$. Fie $M \in \mathbb{R}$. Se caută toate submulțimile B ale lui A pentru care suma elementelor este M . Pentru a putea rezolva problema prin metoda backtracking vom reprezenta

soluția sub forma $\chi = (\chi_1, \dots, \chi_n)$ unde $\chi_i = 1$ dacă $a_i \in B$ și $\chi_i = 0$ în caz contrar. Să luăm $n = 4$. Arborele atașat metodei backtracking este reprezentat mai jos:

| | | | | | | | | |
|----------|---|---|---|---|---|---|---|---|
| χ_1 | 0 | | | | 1 | | | |
| χ_2 | | | | | | | | |
| χ_3 | 0 | | 1 | | 0 | | 1 | |
| χ_4 | | | | | | | | |
| | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
| | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |

Câștigul obținut prin introducerea condițiilor de continuare constă în faptul că dacă într-un vârf ele nu sunt verificate, se renunță la parcurgerea subarborelui care are ca rădăcina acest vârf. Exemplul de mai sus permite prezentarea unei variante a metodei backtracking. Într-adevăr, să considerăm drept soluție posibilă o valoare $k \leq n$ împreună cu k-uplu (χ_1, \dots, χ_k) unde pentru $i \in \{1, \dots, k\}$, χ_i reprezintă indicele elementului pe care îl introducem în B. Evident $\chi_i \neq \chi_j$ pentru $i \neq j$. Pentru a nu repeta soluții, vom presupune că $\chi_1 < \chi_2 < \dots < \chi_k$. Obținem arborele în care fiecare vârf corespunde unei soluții posibile.

| | | | | | | | | | |
|----------|-------|---|---|---|---|-----|--|---|---|
| χ_1 | _____ | 1 | | | | 2 3 | | 4 | |
| χ_2 | _____ | | | | | | | | |
| χ_3 | _____ | 2 | | 3 | 4 | 3 | | 4 | 4 |
| χ_4 | _____ | | | | | | | | |
| | | 3 | 4 | 4 | 4 | | | | |
| | | 4 | | | | | | | |

Diferite variante ale metodei backtracking nu schimbă esența ei, care constă în faptul că este reprezentabilă pe un arbore care este parcurs prin metoda DF, „coborând” în arbore numai dacă există șanse de a ajunge la o soluție rezultat.

Prin urmare, metoda backtracking este, în general, ineficientă având complexitate exponențială. Ea se utilizează totuși în situațiile în care se pot face optimizări în procesul de căutare, evitând, într-o etapă cât mai timpurie a analizei, căile care nu duc la soluție. Există și situații când rezolvarea prin metoda backtracking nu poate fi înlocuită prin alte metode mai eficiente cum ar fi, de exemplu, problemele în care se cer toate soluțiile acceptabile.

2.2.2. Backtracking generalizat (în plan)

Probl.5. Mai sus am aplicat metoda backtracking pentru rezolvarea problemelor în care soluția era reprezentată ca vector. Putem generaliza ideea căutării cu revenire și pentru probleme în care căutarea se face „în plan”, care poate fi reprezentat ca un tablou bidimensional.

Fie un caroiaj având m linii și n coloane. Să presupunem că un mobil (piesa de șah, robot etc.) pleacă din punctul (pătratul) inițial (i_0, j_0) , unde i_0 reprezintă numărul liniei, iar j_0 reprezintă numărul coloanei și că el se deplasează conform unor reguli sintetizate (memorate) în doi vectori d_i și d_j având o dimensiune d dată, cu următoare semnificație: dacă mobil se află în punctul de coordonate (i, j) , el se poate deplasa doar într-unul dintre punctele $(i + d_i[k], j + d_j[k])$, $k=1, 2, \dots, d$, bineînțeles cu condiția ca să nu iasă din caroiaj.

De exemplu, să presupunem că mobilul se poate deplasa doar astfel:

- cu o poziție la dreapta pe aceeași linie;
- cu o poziție la stânga pe aceeași linie;
- cu o poziție în sus pe aceeași coloană;
- cu o poziție în jos pe aceeași coloană;

Ca urmare a acestor posibilități de mișcare a mobilului, vom avea:

$$d_i = (0, 0, 1, -1)$$

$$d_j = (1, -1, 0, 0),$$

astfel că, de exemplu, pentru $k = 2$ vom avea $d_i[k] = 0$ și $d_j[k] = -1$ ceea ce are următoare semnificație: mobilul se deplasează zero linii și o coloană în jos.

În practica apar deseori astfel de situații și, mai mult, în unele probleme, se consideră că anumite pătrate ale caroiajului sunt ocupate (ceea ce înseamnă că mobilul nu poate fi plasat prin deplasări succesive într-un astfel de pătrat).

Frecvent se întâlnesc probleme ce constau în:

- simularea mișcării mobilului;
- determinarea pătratelor în care mobilul poate să ajungă prin deplasări permise;
- determinarea unei succesiuni de deplasări în urma cărora mobilul poate să ajungă într-un punct (if, jf) dat, dacă o astfel de succesiune există;
- determinarea celei mai scurte succesiuni de deplasări în urma cărora mobilul poate să ajungă într-un punct (if, jf) dat, accesibil din punctul inițial (traseu optim).

Astfel de probleme pot fi rezolvate și prin aplicarea metodei backtracking (dar în unele situații sunt mult mai eficiente alte metode cum ar fi, de exemplu, metoda Branch and Bound).

Pentru rezolvarea unor astfel de probleme cu metoda backtracking, vectorului χ întâlnit până acum va avea în continuare o nouă semnificație. Elementele sale vor lua valori în mulțimea 1, 2, ..., d iar semnificația sa este următoarea: dacă după k mutări mobilul a ajuns în poziția (i, j) , atunci, după următoarea sa mutare, mobilul va ajunge în poziția $(i + di[x[k]], j + dj[x[k]])$. Cu alte cuvinte, vectorul χ va conține numărul de ordine al deplasărilor permise (adică va conține indicele la care s-a ajuns în tablourile di și dj).

Deci această tehnică de programare se folosește în rezolvarea problemelor care satisfac următoarele condiții:

- soluția problemei se reprezintă sub forma unui vector $x=(x_1, \dots, x_n)$ unde $x_1 \in A_1, \dots, x_n \in A_n$
- mulțimile A_1, \dots, A_n sunt mulțimi finite și ordonate
- elementele vectorului soluție x trebuie să satisfacă un set de condiții, numite **condiții interne**
- nu se dispune de o altă metodă de rezolvare, mai rapidă.

Observații

- de multe ori nu se cunoaște numărul n al elementelor din soluție, ci o condiție ce trebuie să o satisfacă elementele vectorului x , numită **condiție finală**.
- componentele vectorului x , x_1, \dots, x_n pot fi la rândul lor vectori
- în cele mai multe situații, mulțimile A_1, \dots, A_n coincid
- problemele care se pot rezolva utilizând această tehnică sunt cele așa numite **nedeterminist polinomiale**, pentru care nu se cunoaște o soluție iterativă.

Principiul de generare a soluției

- soluția se construiește pas cu pas: x_1, x_2, \dots, x_n
- presupunând generate la un moment dat elementele x_1, x_2, \dots, x_k , aparținând mulțimilor A_1, \dots, A_k se alege (dacă există) x_{k+1} , primul element neales încă din mulțimea A_{k+1} și care verifică condițiile de continuare, altfel spus subsoluția x_1, x_2, \dots, x_{k+1} să satisfacă **condițiile interne**. Apar două posibilități:
 1. nu s-a găsit un astfel de element, se revine, considerând generată subsoluția x_1, \dots, x_{k-1} și se caută o altă valoare pentru x_k - următorul element netestat încă din mulțimea A_k .
 2. s-a găsit o valoare corespunzătoare pentru x_{k+1} , caz în care apar din nou două situații:
 - i. s-a ajuns la soluție, se tipărește soluția și se reia algoritmul considerând generată subsoluția x_1, \dots, x_k (se caută pentru x_{k+1} un element din mulțimea A_{k+1} rămas netestat)
 - ii. nu s-a ajuns încă la soluție, caz în care se consideră generată subsoluția x_1, \dots, x_{k+1} și se caută un prim element $x_{k+2} \in A_{k+2}$.

Algoritmul se termină atunci când au fost testate toate elementele din A_1 .

Observații

- în urma aplicării tehnicii backtracking, se generează toate soluțiile problemei. În cazul în care se cere doar o singură soluție, se poate opri generarea atunci când a fost găsită
- alegerea condițiilor de continuare este foarte importantă, deoarece duce la micșorarea numărului de calcule
- problemele rezolvate folosind tehnica backtracking necesită un timp de execuție îndelungat. Spre exemplu, presupunând că $A_1 = A_2 = \dots = A_n = \{1, 2, \dots, n\}$, numărul total al subsoluțiilor care se pot obține - fără a lua în considerare condițiile interne - este n^n . Chiar dacă nu se poate evalua exact - depinde de condițiile interne alese -, timpul de execuție crește exponențial, pe măsura creșterii lui n .

În continuare vom prezenta variante ale tehnicii backtracking și o serie de probleme rezolvate prin această metodă.

2.2 BACKTRACKING SIMPLU (ELEMENTAR)

2.2.1 VARIANTA NERECURSIVĂ

Probl.6. În continuare vom prezenta subprogramul “backtracking”, care generează toate soluțiile problemei. Am presupus următoarele:

- soluția problemei este vectorul x_1, \dots, x_k , ale cărei elemente satisfac o **condiție finală**, verificată cu ajutorul funcției FINAL
- $A_k = \{1, 2, \dots, c[k]\}$, $\forall k \in \{1, \dots, n\}$
- **condițiile interne** sunt verificate în funcția POSIBIL
- marcarea elementelor din A_k nealese se face dându-i lui $x[k]$ valoarea 0
- procedura SCRIE tipărește o soluție a problemei

```
void backtracking(void)
{
    k=1;
    x[k]=0;
    while (k)
    {
        înainte=0;
        while(!înainte && (x[k] < c[k]))
        {
            x[k]++;
            înainte=posibil(k);
        }
        if (înainte)
            if (final(k))
                scrie();
            else
                x[++k]=0;
            else --k;
    }
}
```

2.2.1.1 Generarea permutărilor

Probl.7. Dându-se șirul a_1, \dots, a_n se cere să se genereze toate permutările elementelor șirului
Algoritmul folosit este BACKTRACKING.

Vom genera de fapt permutările mulțimii $\{1, 2, \dots, n\}$ - reprezentând indicii elementelor din șir.

- soluția este x_1, \dots, x_n - **condiția finală** : $k=n$
- **condiția internă** : pentru ca x_k să fie valid, trebuie să fie diferit de elementele generate înaintea sa: x_1, \dots, x_{k-1} adică $x_k \neq x_i, \forall i \in \{1, \dots, k-1\}$

Observații:

1. Datele de intrare se citesc dintr-un fișier text
2. În variabila SOL contorizăm numărul de soluții ale problemei (numărul de permutări)

```
#include<stdio.h>
#include<conio.h>
#include<iostream.h>
int n,a[20],x[20],sol=0;
/*****/
void citire(void)
{
    int i;
    FILE *pf=fopen("fis.txt","r");
    fscanf(pf,"%d",&n);
    for(i=1;i<=n;i++)    fscanf(pf,"%d",&a[i]);
    fclose(pf);
}
/*****/
int posibil(int k)
{
    int l;
    for (l=1;l<=k-1;l++)    if (x[k]==x[l]) return 0;
    return 1;
}
/*****/
void scrie(void)
{
    int i; sol++; printf("\n");
    for(i=1;i<=n;i++)    cout<<a[i]<<" ";
    getch();
}
```

```

    }

    /***/
    void back(void)
    { int k,inainte; k=1; x[k]=0;
      while (k)
      {
        inainte=0;
        while(!inainte&&(x[k]<n))
        {
          x[k]++;
          inainte=posibil(k);
        }
        if (inainte)
          if (k==n)
            scrie();
          else
            x[++k]=0;
          else --k;
        }
      }

    /***/
    void main(void)
    {
      citire();
      clrscr();
      back();
      cout<<endl<<"Sunt " <<sol<<" permutări";
      getch();
    }

```

Observație

Generarea permutărilor folosind metoda **backtracking** nu este deloc eficientă, mai ales din punctul de vedere al timpului necesar execuției. Am prezentat rezolvarea anterioară doar pentru a ilustra metoda.

Vom propune în continuare un algoritm de generare al permutărilor, al cărui ordin de complexitate este $O(n^2)$, fiind mult mai eficient decât algoritmul backtracking.

Problema 8. Să se genereze permutările mulțimii $\{1, \dots, n\}$.

Notatii

- p_1, \dots, p_n vectorul în care se reține la un moment dat o permutare.
- np numărul final de permutări
- **SCRIE** - procedura care tipărește o permutare

Algoritmul este următorul:

1. se inițializează permutarea p cu permutarea identică; se inițializează k cu 1 - k reprezintă poziția elementului curent prelucrat (ideea va fi ca pe poziția k să se aducă pe rând toate valorile posibile)
2. până când k ajunge egal cu n se execută
 - iii. se reține valoarea v a lui $p[k]$, elementele $p[k+1], \dots$, până se deplasează cu o poziție în stânga, iar pe poziția n se pune valoarea v
 - iv. dacă $p[k]=k$, înseamnă că pe poziția k au ajuns toate valorile posibile, deci se poate trece la prelucrarea următorului element (se incrementează k); se trece la pasul 2
 - v. dacă $p[k] \neq k$, înseamnă că s-a obținut o permutare, care se va tipări; se reinițializează poziția curentă cu 1 și se trece la pasul 2

```

#include<stdio.h>
#include<conio.h>
#include<iostream.h>
int *p,n,np;

/***/
void scrie(void)
{
  np++;
  cout<<"Permutarea nr: " <<np<<endl;
  for(int i=1;i<=n;i++)
    cout<<p[i]<<" ";
  cout<<endl;
}

```

```

    getch();
}

/*****
void main(void)
{
    clrscr();
    cout<<"n=";
    cin>>n;
    p=new int[n+1];
    for (int i=1;i<=n;i++)
        p[i]=i;
    cout<<"Permutarea nr: 1"<<endl;
    for(int i=1;i<=n;i++)
        cout<<p[i]<<" ";
    cout<<endl;
    int k=1;
    np=1;
    do
    {
        int aux=p[k];
        for (int i=k;i<n;i++)
            p[i]=p[i+1];
        p[n]=aux;
        if (p[k]==k) k++;
        else
        {
            scrie();
            k=1;
        }
    }
    while(k<=n-1);
    cout<<"Sunt: "<<np<<" permutări";
    delete p;
    getch();
}

```

2.2.2 VARIANTA RECURSIVĂ. În continuare vom prezenta subprogramul recursiv “backtracking”, care generează toate soluțiile problemei.

Problema 9. Am presupus următoarele

- soluția problemei este vectorul x_1, \dots, x_k , ale cărui elemente satisfac o **condiție finală**, verificată cu ajutorul funcției FINAL
- $A_k = \{1, 2, \dots, c[k]\}$, $\forall k \in \{1, \dots, n\}$
- **condițiile interne** sunt verificate în funcția POSIBIL
- procedura SCRIE tipărește o soluție a problemei

```

void back(int k)
{ for(int l=1;l<=c[k];l++)
  { x[k]=l;
    if (posibil(k))
      if (final(k)) scrie();
      else back(k+1);
  }
}

```

Observație:

Dacă se dorește generarea unei singure soluții, care, de exemplu satisface o condiție de optim - maxim sau minim -, se va înlocui procedura SCRIE cu procedura COMPARA, care compară soluția curentă cu o soluție optimă, care va fi tipărită în final.

2.2.2.1 Generarea combinărilor.

Problema 10. Dându-se n și p numere naturale ($n \geq p$), se cere să se genereze toate submulțimile cu p elemente ale mulțimii $\{1, \dots, n\}$. Două submulțimi se consideră egale dacă și numai dacă au aceleași elemente, indiferent de ordinea în care acestea apar.

Algoritmul folosit este BACKTRACKING.

- soluția este x_1, \dots, x_p - **condiția finală** : $k=p$

- $A_k = \{1, 2, \dots, n\}$, $\forall k \in \{1, \dots, n\}$
- **condițiile interne** : pentru ca x_k să fie valid, trebuie ca:
 - să fie diferit de elementele generate înaintea sa: x_1, \dots, x_{k-1} adică $x_k \neq x_i$, $\forall i \in \{1, \dots, k-1\}$
 - $x_k > x_{k-1}$ - pentru a evita generarea unei submulțimi cu aceleași elemente de mai multe ori. De fapt condiția a doua este suficientă.

Observații:

1. Datele de intrare se citesc dintr-un fișier text
2. În variabila SOL contorizăm numărul de soluții ale problemei (numărul de combinații)

```
#include <stdio.h>
#include <conio.h>
#include <iostream.h>
int n,p,x[20],sol=0;
//*****
void citire(void)
{ FILE *pf=fopen("fis.txt","r");
  fscanf(pf,"%d %d",&n,&p);
  fclose(pf);
}
//*****
int posibil(int k)
{ if (k>1)
  if (x[k]<=x[k-1]) return 0;
  return 1;
}
//*****
void scrie(void)
{ sol++;
  cout<<"\n";
  for(int i=1;i<=p;i++)
    cout<<x[i]<<" ";
}
//*****
void back(int k)
{
  for(int l=1;l<=n;l++)
  {
    x[k]=l;
    if (posibil(k))
      if (k==p) scrie();
      else back(k+1);
  }
}
//*****
void main(void)
{
  citire();
  clrscr();
  back(1);
  cout<<endl<<"Sunt "<<sol<<" combinații";
  getch();
}
```

2.2.2.2 Subșir crescător de lungime maximă

Problema 11. Se dă șirul a_1, \dots, a_n . Se cere să se genereze subșirul crescător de lungime maximă, adică o succesiune de indici x_1, \dots, x_k astfel încât $x_1 < x_2 < \dots < x_k$ și în plus $a[x_1] < a[x_2] < \dots < a[x_k]$

Algoritmul folosit este BACKTRACKING.

- de data aceasta se cere o singură soluție, aceea de lungime maximă
- condițiile interne sunt specificate în enunțul problemei

Observații:

1. Datele de intrare se citesc dintr-un fișier text
2. Notății: x_{max} - vectorul ce reprezintă soluția maximă
 k_{max} - dimensiunea soluției maxime

COMPARA (k) - procedura care compară soluția curentă
cu soluția maximă

SCRIE - procedura care tipărește soluția maximă

3. Folosirea metodei BACKTRACKING în acest caz, nu este indicată, deoarece se cunoaște o metodă polinomială pentru rezolvarea acestei probleme (metoda programării dinamice, care va fi prezentată în capitolul 5). Am descris-o doar pentru a ilustra modul în care se poate genera o soluție ce satisface o condiție de optim.

```
#include <stdio.h>
#include <conio.h>
#include <iostream.h>
int n,kmax=0,a[20],xmax[20],x[20];
//*****
void citire(void)
{
    int i;
    FILE *pf=fopen("fis.txt","r");
    fscanf(pf,"%d",&n);
    for(i=1;i<=n;i++)
        fscanf(pf,"%d",&a[i]);
    fclose(pf);
}

//*****
int posibil(int k)
{
    if (k>1)
    {
        if (x[k]<=x[k-1]) return 0;
        if (a[x[k]]<=a[x[k-1]]) return 0;
    }
    return 1;
}

//*****
void compara(int k)
{
    if (k>kmax)
    {
        kmax=k;
        for (int i=1;i<=k;i++)
            xmax[i]=x[i];
    }
}

//*****
void scrie(void)
{
    cout<<"\n";
    for(int i=1;i<=kmax;i++)
        cout<<a[xmax[i]]<<" ";
}

//*****
void back(int k)
{
    for(int l=1;l<=n;l++)
    {
        x[k]=l;
        if (posibil(k))
        {
            compara(k);
            back(k+1);
        }
    }
}

//*****
```

```

void main(void)
{
    citire();
    clrscr();
    back(1);
    cout<<endl<<"Subșirul crescător de lungime maxima este:\n";
    serie();
    getch();
}

```

Observație În general se încearcă evitarea metodei backtracking, mai ales dacă se găsește un algoritm polinomial de rezolvare al problemei.

Propunem în continuare următoarea problemă

Problema 12. *La un magazin de dulciuri se găsesc spre vânzare $n-1$ tipuri de ciocolate, având prețurile p_1, \dots, p_{n-1} . Nici una din ciocolate nu poate fi cumpărată prin plata exactă cu bancnote de valoare n . Un copil vine la magazin și vrea să cumpere ciocolate. El are la dispoziție un număr nelimitat de bancnote de valoare n și o bancnotă de valoare m . Să se precizeze care sunt ciocolatele pe care trebuie să le cumpere copilul, pentru a putea plăti exact suma, știind că nici o ciocolată nu are prețul m .*

Problema se reduce de fapt la următoarea: Dându-se șirul p_1, \dots, p_{n-1} și o valoare m și știind că elementele șirului nu sunt divizibile cu n și sunt diferite de m se cere să se determine o submulțime a șirului astfel încât suma elementelor din această submulțime să dea restul 0 sau m la împărțirea cu n .

La o analiză superficială, s-ar părea că este vorba de un **backtracking**, generarea unei submulțimi a unui șir, ce satisface condiția impusă de problemă.

La o analiză mai atentă, se poate găsi un algoritm liniar ce rezolvă problema. Pentru aceasta, se utilizează **Principiul cutiei lui Dirichlet** care s-ar enunța astfel (într-o formulare mai simplă) “Dându-se p obiecte și $p-1$ cutii și știind că fiecare obiect trebuie introdus într-o cutie, vor exista cel puțin 2 obiecte ce vor fi introduse în aceeași cutie”.

Algoritmul liniar pentru rezolvarea problemei este:

1. se construiește șirul sumelor parțiale formate cu elementele șirului p , adică $s_1=p_1$, $s_2=p_1+p_2$, ..., $s_{n-1}=p_1+p_2+\dots+p_{n-1}$.
2. se caută în șirul s un element care dă restul 0 sau m la împărțirea cu n
 - există un astfel de element, și, atunci problema este rezolvată, s-a găsit soluția, adică subșirul **p_1, p_2, \dots, p_i**
 - nu s-a găsit nici un element, ceea ce înseamnă că elementele s_1, \dots, s_{n-1} dau prin împărțirea la n resturile $\{1, 2, \dots, n-1\} - \{m\}$.
 - Dacă există în șirul s 2 elemente s_i și s_j ($j>i$) astfel încât restul împărțirii lui $s_j - s_i$ la n să fie m , am găsit soluția problemei ca fiind subșirul **p_{i+1}, \dots, p_j** (deoarece $s_j - s_i = p_{i+1} + \dots + p_j$)

Dacă nu există în șirul s 2 elemente s_i și s_j ($j>i$) astfel încât restul împărțirii lui $s_j - s_i$ la n să fie m , aplicăm principiul cutiei lui Dirichlet, ceea ce înseamnă că există 2 elemente s_i și s_j ($j>i$) care dau același rest la împărțirea cu n . Deci valoarea $s_j - s_i$ este multiplă de n și din nou am găsit soluția problemei ca fiind subșirul **p_{i+1}, \dots, p_j** (deoarece $s_j - s_i = p_{i+1} + \dots + p_j$)

Observații

- ordinul de complexitate al algoritmului este $O(n)$, adică algoritmul este liniar.
- algoritmul descris anterior ne asigură că există o succesiune de elemente consecutive în șir a căror sumă dă restul 0 sau m la împărțirea cu n , ceea ce înseamnă că se evită rezolvarea cu backtracking, care ar necesita timp mare de execuție.

```

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
int n,q,p[100],s[100];
/*****
void citire(void)
{
    int i;
    FILE *pf;
    pf=fopen("fis.txt","r");
    fscanf(pf,"%d %d",&n,&q);
    s[0]=0;
    for(i=1;i<=n-1;i++)
    {

```

```

        fscanf(pf,"%d",&p[i]);
        s[i]=s[i-1]+p[i];
    }
    fclose(pf);
}

/*****/
void scrie(int s,int d)
{
    int i;
    for(i=s;i<=d;i++)
        printf("%d ",p[i]);
    getch();
}

/*****/
void alegere(void)
{
    int i,j,v;
    clrscr();
    printf("Elementele sunt:\n");
    for(i=1;i<=n-1;i++)
        if (!(s[i]!n)||(s[i]!n==q))
        {
            scrie(1,i);
            exit(1);
        }
    for(i=1;i<=n-2;i++)
        for(j=i+1;j<=n-1;j++)
        {
            v=(s[j]-s[i])!n;
            if (!(v||(v==q))
            {
                scrie(i+1,j);
                exit(1);
            }
        }
}

/*****/
void main(void)
{
    citire();
    alegere();
}

```

Problema 13. Problema “matricelor”. Sa se genereze toate matricele $n \times n$ ce conțin elemente distincte din mulțimea $\{1, \dots, n^2\}$ astfel încât pe fiecare linie elementele sa fie în ordine crescătoare.

Soluția folosită se bazează pe metoda **backtracking** și este realizată recursiv. Elementele se introduc în matrice crescător pe linii, iar în cadrul liniei, crescător după coloana, cu ajutorul procedurii recursive BACK(i,j). Funcția POSIBIL(i,j) verifică dacă elementul din matrice de pe poziția i,j corespunde condițiilor impuse de problema.

Observație: Datorită metodei de rezolvare, **backtracking**, timpul de execuție este exponențial

```

#include <stdio.h>
#include <conio.h>
#include <iostream.h>
int n,a[20][20];

/*****/
int posibil(int i,int j)
{
    int l;
    for(int k=1;k<=i-1;k++)
        for(l=1;l<=n;l++) if (a[i][j]==a[k][l]) return 0; if ((j) && (a[i][j]<=a[i][j-1])) return 0; return 1; }
/*****/
void tiparire(void)
{
    cout<<"\n";
    for(int i=1;i<=n;i++) { cout<<"\n";
        for(int j=1;j<=n;j++) cout<<a[i][j]<<" "; } getch(); }

```



```

//*****
void back(int i,int j)
{   for(int k=1;k<=n*n;k++) { a[i][j]=k;
    if (posibil(i,j))
        if ((i==n) && (j==n)) tiparire();
        else if (j==n) back(i+1,1);    else back(i,j+1);    } }
//*****

void main(void)
{ n=2;   clrscr(); back(1,1); getch(); }

```

3. BACKTRACKING ÎN PLAN (BIDIMENSIONAL)

La problemele prezentate în secțiunea anterioară, soluția se reprezenta sub forma unui vector, altfel spus stiva soluție avea lățimea 1.

Există probleme pentru care se folosește stiva dublă, triplă, adică pe fiecare nivel în stivă trebuie reținute mai multe componente. Vom conveni să numim **backtracking în plan** metoda folosită pentru rezolvarea problemelor în care se cere determinarea unui drum în plan - pentru fiecare poziție se vor reține două valori, atât abscisa cât și ordonata.

Rutina corespunzătoare poate fi descrisă astfel:

```

void back(int k)
{   for(int l=1;l<=c[k];l++)
    {   // determina poziția corespunzătoare etapei k
        if (posibil(k))
            if (final(k)) scrie();
            else back(k+1);
    }
}

```

- am presupus că la etapa k avem c[k] variante de deplasare.
- **condițiile interne** sunt verificate în funcția POSIBIL
- **condiția finală** este verificată în funcția FINAL
- procedura SCRIE tipărește o soluție a problemei

Observații: ca și în cazul backtrackingului simplu, se poate cere generarea unei singure soluții, sau a unei soluții optime. În acest caz se va proceda ca și în cazul backtrackingului elementar.

3.1 BACKTRACKING FĂRĂ MODIFICAREA CONFIGURAȚIEI

Problema 14. Problema “labirintului”

Un labirint este reprezentat sub forma unei matrice de dimensiune nxn, având nxn camere. Anumite camere sunt ocupate, altele sunt libere. Se cere să se genereze toate traseele unei persoane, care intră în labirint pe prima linie a acestuia și iese pe ultima linie, știind că:

- nu poate trece de două ori prin aceeași poziție
- poate trece doar prin poziții libere
- dintr-o anumită poziție, se poate deplasa în patru direcții (N, S, E, V)

Algoritmul folosit este BACKTRACKING.

Labirintul va fi reprezentat sub forma unei matrice binare A(nxn), un element $A_{i,j}$ fiind egal cu 1 dacă poziția este liberă, respectiv 0 dacă poziția este ocupată. Problema se reduce la generarea unui șir de poziții (linie, coloană) prin care trebuie să treacă persoana, respectând condițiile impuse.

- soluția va fi reprezentată sub forma unei matrice cu 2 linii $(y_{1,1} - y_{1,2})$, $(y_{2,1} - y_{2,2})$, ..., $(y_{k,1} - y_{k,2})$ cu semnificația:

$y_{i,1}$ - linia pe care se află persoana la etapa i

$y_{i,2}$ - coloana pe care se află persoana la etapa i

• **condiția finală** : $y_{k,1} = n$ (se ajunge pe ultima linie a labirintului)

• **condițiile interne** : pentru ca poziția la etapa k $(y_{k,1} - y_{k,2})$ să fie validă, trebuie ca:

- să fie diferită de toate pozițiile prin care s-a trecut la etapele anterioare adică $(y_{k,1} - y_{k,2}) \neq (y_{i,1} - y_{i,2})$, $\forall i \in \{1, \dots, k-1\}$
- să nu se treacă decât prin poziții libere, adică elementul din matricea A de pe poziția $(y_{k,1}, y_{k,2})$ să fie egal cu 1.
- pentru deplasarea persoanei, cele 4 direcții le vom simboliza astfel: 1 - nord, 2 - est, 3 - sud, 4 - vest. Vom utiliza doi vectori dx și dy pentru a ilustra deplasarea relativă în cele 4 direcții (față de poziția

curentă). Astfel, dacă poziția curentă în matrice este (i,j), atunci poziția corespunzătoare vecinului din direcția k va fi (i+dx_k , j+dy_k)

Observații:

1. Datele de intrare se citesc dintr-un fișier text
2. În variabila SOL contorizăm numărul de soluții ale problemei (numărul de posibil de trasee)
3. Folosim următoarele subprograme

POSIBIL(k) - funcția care verifică dacă poziția (y_{k,1} , y_{k,2}) este validă

SCRIE(k) - procedura care tipărește o soluție

BACK(k) - procedura care generează recursiv soluția

```
#include <stdio.h>
#include <conio.h>
#include <iostream.h>
int n,a[20][20],y[20][3],sol=0;
int dx[4]={-1,0,1,0};
int dy[4]={0,1,0,-1};

//*****

void citire(void)
{
    FILE *pf=fopen("mat.txt","r");
    fscanf(pf,"%d",&n);
    for(int i=1;i<=n;i++)
        for(int j=1;j<=n;j++)
            fscanf(pf,"%d",&a[i][j]);
    fclose(pf);
}

//*****

int posibil(int k)
{
    if ((y[k][1]<1)||((y[k][1]>n)||((y[k][2]<1)||((y[k][2]>n)) return 0;
    if (!a[y[k][1]][y[k][2]]) return 0;
    for (int l=1;l<=4;l++)
        if ((y[k][1]==y[l][1])&&(y[k][2]==y[l][2])) return 0;
    return 1;
}

//*****

void scrie(int k)
{
    sol++;
    cout<<"\n Soluția " << sol << "\n\n";
    for(int i=1;i<=k;i++)
        cout<<y[i][1]<<"-"<<y[i][2]<<" ";
}

//*****

void back(int k)
{
    for(int l=0;l<4;l++)
    {
        y[k][1]=y[k-1][1]+dx[l];
        y[k][2]=y[k-1][2]+dy[l];
        if (posibil(k))
            if (y[k][1]==n) scrie(k);
            else back(k+1);
    }
}

//*****

void main(void)
{
    citire();
    clrscr();
    for(int j=1;j<=n;j++)
        if (a[1][j])
        {
```

```

        y[1][1]=1;
        y[1][2]=j;
        back(2);
    }
    if (!sol) cout<<"Nu exista solutii";
    getch();
}

```

3.2 BACKTRACKING CU MODIFICAREA CONFIGURAȚIEI

Spunem că este vorba de un *backtracking în plan cu modificarea configurației* în cazul problemelor de următorul tip: vrem să generăm trasee în plan (pe o matrice, de exemplu), dar la fiecare deplasare, configurația se modifică - altfel spus matricea pe care se face deplasarea se modifică la fiecare etapă. Problema este că după ce la un moment dat s-a ales o variantă de deplasare (și s-a modificat corespunzător configurația) care nu conduce la soluție, la revenirea din recursivitate trebuie refăcută configurația anterioară modificării. Dacă nu se reface corect configurația, se vor obține rezultate eronate. Tot ceea ce a fost prezentat în secțiunea anterioară, cu observația că trebuie făcută o modificare a rutinei care genera recursiv soluțiile problemei.

Rutina corespunzătoare poate fi descrisă astfel:

```

void back(int k)
{
    for(int l=1;l<=c[k];l++)
    {
        // determina poziția corespunzătoare etapei k
        if (posibil(k))
        {
            // retine configurația curenta CONFIG:=CURENT
            // modifica configurația curenta
            if (final(k)) scrie();
            else back(k+1);
            //refă configurația curenta CURENT:=CONFIG
        }
    }
}

```

Problema 15. Problema generalizată a "labirintului"

Un labirint este reprezentat sub forma unei matrice de dimensiune $n \times n$, având $n \times n$ camere. Anumite camere sunt ocupate, altele sunt libere. Se cere să se genereze toate traseele unei persoane, care intră în labirint pe prima linie a acestuia și iese pe ultima linie, știind că:

- nu poate trece de două ori prin aceeași poziție
- poate trece doar prin poziții libere
- dintr-o anumită poziție, se poate deplasa în patru direcții (N, S, E, V)
- la trecerea printr-o poziție a labirintului, pozițiile vecine își schimbă starea (pozițiile ocupate devin libere, iar cele libere devin ocupate)

Algoritmul folosit este BACKTRACKING CU MODIFICAREA CONFIGURAȚIEI - ultima condiție sugerează modificarea configurației la fiecare etapă.

În rest, modul de abordare al problemei este același ca la problema labirintului (2.3.1.1)

```

#include <stdio.h>
#include <conio.h>
#include <iostream.h>
int n,a[20][20],y[20][3],sol=0;
int dx[4]={-1,0,1,0};
int dy[4]={0,1,0,-1};
//*****
void citire(void)
{
    FILE *pf=fopen("mat.txt","r");
    fscanf(pf,"%d",&n);
    for(int i=1;i<=n;i++)
        for(int j=1;j<=n;j++)
            fscanf(pf,"%d",&a[i][j]);
    fclose(pf);
}
//*****
int posibil(int k)

```

```

{
    if ((y[kt[1t<1])||(y[kt[1t>n])||(y[kt[2t<1])||(y[kt[2t>n])) return 0;
    if (!a[y[kt[1tt][y[kt[2tt] return 0;
    for (int l=1;l<=k-1;l++)
        if ((y[kt[1t==y[l][1t]&&(y[kt[2t==y[l][2t)) return 0;
    return 1;
}

//*****
void scrie(int k)
{
    sol++;
    cout<<"\n Soluția "<<sol<<"\n\n";
    for(int i=1;i<=k;i++)
        cout<<y[i][1t<<" "<<y[i][2t<<" ";
}

//*****
void modif(int k)
{
    for (int i=0;i<4;i++)
        if ((y[kt[1t+dx[it]>=1) && (y[kt[1t+dx[it]<=n) && (y[kt[2t+dy[it]>=1) &&
        (y[kt[2t+dy[it]<=n))
            a[y[kt[1t+dx[i][y[kt[2t+dy[i]=1;
}

//*****
void back(int k)
{
    int d[20t[20t,i,j,l;
    for(i=0;i<4;i++)
    {
        y[kt[1t=y[k-1t[1t+dx[i;
        y[kt[2t=y[k-1t[2t+dy[i;
        if (posibil(k))
        {
            //Se retine configurația înainte de modificare
            for(j=1;j<=n;j++)
                for(l=1;l<=n;l++)
                    d[jt[l]=a[jt[l];
            //Se efectuează modificarea configurației curente A
            modif(k);
            if (y[kt[1t==n) scrie(k);
            else back(k+1);
            //Se reface configurația de dinaintea modificării
            for(j=1;j<=n;j++)
                for(l=1;l<=n;l++)
                    a[jt[l]=d[jt[l];
        }
    }
}

//*****
void main(void)
{
    citire();
    clrscr();
    for(int j=1;j<=n;j++)
        if (a[1t[jt]
        {
            y[1t[1t=1;
            y[1t[2t=j;
            modif(1);
            back(2);
        }
    if (!sol) cout<<"Nu exista soluții";
    getch();
}

```

}

Problema 16. Problema colorării hărților cu r culori

Fiind dată o hartă să se coloreze țările cu cele r culori astfel încât fiecare țară să fie colorată cu o singură culoare și să nu existe două țări vecine colorate cu aceeași culoare.

R. Aplicăm metoda backtracking, considerând soluția un vector $c=(c_1, c_2, \dots, c_n)$ unde n este numărul de țări iar c_i este culoarea cu care este colorată țara i , deci fie $\{1, 2, \dots, r\}$.

Algoritmul de colorare este următorul:

```
Program Colorare( $n, r, H, c$ )
  array  $c(n), H(n, n)$ .
   $k:=1$ ;  $ck:=0$ 
  while  $k>0$ 
     $v:=0$ 
    while  $ck<r$ 
       $ck:=ck+1$ 
      call Posibil( $c, k, H, v$ )
      if  $v=1$  then exit
    endif
  repeat
    if  $v=0$  then  $k:=k-1$ 
  else if  $k=n$  then call ScrieRez( $c, n$ )
  else  $k:=k+1$ ;  $ck:=0$ 
  endif
endif
repeat
end
```

Procedure Posibil(c, k, H, v) **array** $c(n), H(n, n)$

for $i=1, k-1$

if $ck=c_i$ and $H(i, k)=1$ **then** $v:=0$; **return**

endif

repeat

$v:=1$

return

end

Procedura *ScrieRez* afișează valorile vectorului c , adică culorile cu care sunt colorate țările, iar matricea H indică prin elementele sale configurația hărții, $H(i, j)=1$ dacă țara i este vecină cu țara j și 0 altfel.

Algoritmul furnizează toate posibilitățile de colorare a hărții cu cele r culori.

Problema 17. Problema celor opt dame. Să se genereze toate așezările posibile a opt dame pe tabla de șah astfel încât acestea să nu se atace reciproc (două dame se atacă reciproc dacă sunt situate pe aceeași linie, coloană sau pe diagonală).

Să se generalizeze problema pentru n dame pe o tablă de $n \times n$.

R. Aplicăm metoda backtracking considerând soluția un vector $x=(x_1, x_2, \dots, x_n)$ unde x_i , aparține $\{1, 2, \dots, n\}$ și reprezintă numărul coloanei în care este așezată dama i .

Cu ajutorul procedurii *Posibil* verificăm dacă dama k este așezată pe aceeași coloană sau diagonală cu o altă damă așezată anterior.

Procedure Posibil(x, k, v)

array $x(n)$

for $i=1, k-1$

if $x_k=x_i$ or $k-i=|x_k-x_i|$ **then** $v:=0$

return

endif

repeat

$v:=1$

return

end

Programul principal este următorul:

Program Dame(n, x)

array $x(n)$

$k:=1$

$x_k:=0$

while $k>0$ $v:=0$

while $x_k<n$

```

xk:=xk+1
call Posibil(x,k;v)
if v=1 then exit
endif
repeat
if v=0 then k:=k-1
else if k=n then call ScrieRez(x,n)
else k:=k+1
xk:=0
endif
endif
repeat
end

```

Problema 18. Săritura calului Pe o tablă de șah să se parcurgă pornind dintr-un colț cu ajutorul unui cal toate pătratele tablei de șah fără a trece de două ori prin același pătrat.

Generalizare pentru o tablă de $n \times n$ pătrate.

R. Se folosește metoda backtracking varianta recursivă, organizând tabla de șah ca o matrice de întregi de dimensiune $n \times n$, cu elemente ce pot lua valorile 0 sau numărul de ordine al mutării ce s-a efectuat în acea poziție. Valoarea 0 atribuită unei poziții de pe tabla de șah semnifică faptul că acea poziție nu a fost ocupată. Algoritmul se termină în momentul în care toate pozițiile de pe tabla de șah au fost ocupate. Acest lucru este testat printr-o variabilă booleană poziționată corespunzător. Se pornește din linia 1 și coloana 1, iar condițiile de continuare la un moment dat din linia x și coloana y sunt următoarele:

$1 \leq xu = x + a_k \leq n, 1 \leq yu = y + b_k \leq n, k=1,8$ unde xu, yu sunt linia și coloana unde se încearcă o nouă mutare, iar a și b sunt doi vectori în care sunt memorate posibilitățile de deplasare ale calului pe tabla de șah.

Algoritmul scris în Pascal este următorul:

```

Program Calut;
uses crt;
const
  Liber=0;
  Nmax=8;
  Succes=1;
  Esec=0;
var
  h:array[1..Nmax,1..Nmax] of integer;
  a,b:array[1..8] of integer;
  n,Np,Timp:integer;
Procedure Afisare;
Var      i,j:integer;
begin
  clrscr;      writeln;
for i:=1 to n do
begin
for j:=1 to n do
if h[i,j]<>Liber then write(' ',h[i,j]:2,' ')
else write(' ':6);
  writeln;writeln
end
end;
Function Incerc(i,x,y:integer):integer;
Var      yu,xu,k,q:integer;
begin
  k:=1;q:=Esec;
while (q=Esec) and (k=8) do
begin
  xu:=x+a[k];
  yu:=y+b[k];
if (xu>=1) and (xu<=N) and (yu>=1) and (yu<=n) then
if h[xu,yu]=Liber then
begin
  h[xu,yu]:=i;
  afisare;

```

```

        delay(Timp);
        if i<Np then
            begin
                q:=incerc(i+1,xu,yu);
                if q=Esec then
                    h[xu,yu]:=Liber
                end
                else q:=Succes
            end;
            k:=k+1
        end;
    incerc:=q
end;
var      i,j:integer;
begin
    write('Dimensiunea tablei (<=8): ');readln(n);
    write('Timpul de așteptare pentru afișarea unei mutări (in microsecunde): ');      readln(Timp);
    Np:=n*n;
    for i:=1 to n do
        for j:=1 to n do      h[i,j]:=Liber;
            a[1]:=2; a[2]:=1;a[3]:=-1;a[4]:=-2;a[5]:=-2;a[6]:=-1;a[7]:=1;a[8]:=2;
            b[1]:=1;b[2]:=2;b[3]:=2;b[4]:=1;b[5]:=-1; b[6]:=-2;b[7]:=-2;b[8]:=-1; h[1,1]:=1;
        if incerc(2,1)=Succes then afisare
            else writeln('Nu exista soluții !!!')
        end.

```

Problema 19. Generare permutări, aranjamente, combinări. Să se genereze permutările mulțimii $\{1,2,\dots,n\}$.

R. Folosim metoda backtracking definind o permutare ca fiind un vector $X=(x_1,x_2,\dots,x_n)$ unde $x_i \in \{1,2,\dots,n\}$ cu condiția ca $x_i < x_j$ ($\forall i < j$), iar $i, j \in \{1,2,\dots,n\}$. Observăm că mulțimile $S_i = \{1,2,\dots,n\}$ ce compun spațiul soluțiilor sunt progresii aritmetice cu primul termen 1, rația 1 și ultimul termen n.

Condițiile de continuare în cazul acesta sunt ca $x_k < x_i$, ($\forall i=1, k-1$).

Algoritmul este:

```

Program Perm(x.n)
array x(n)
k:=1,x1:=0
while k>0
    v:=0
    while xk<=n-1
        xk:=xk+1
        call Posibil(x,k,n,v)
        if v=1 then exit
    endif
    repeat
        if v=0 then k:=0k-1
        else if k=n then call Scrie(x,n)
        else k:=k+1
        xk=0
    endif
    endif
    repeat
    end

Procedure Posibil(x,k,n,v)
array x(n)
for i=1,k-1
    if x(i)=x(k) then v=0
    return
    endif
    repeat
    v:=1
    return
    end

Procedure Scrie(x,n)

```

```

array x(n)
for i=1,n:
write x(i)
repeat
return
end

```

Să se genereze aranjamentele mulțimii $\{1,2,\dots,n\}$ luate câte p elemente, $p \leq n$.

R. Algoritmul este identic cu cel de la permutări cu deosebirea că *if*-ul ce conține scrierea devine:

```

    if k=p then call Scribe(x,p)
    else ...
endif

```

Să se genereze combinările mulțimii $\{1,2,\dots,n\}$ luate câte p elemente, $p \leq n$.

R. Algoritmul este același ca la aranjamente, înlocuindu-se procedura *Posibil* prin:

```

Procedure Posibil(x,k,n,v)
array x(n)
if k=1 then v:=1
return
endif
if xk>xk-1 then v:=1
else v:=0
endif
return
end

```

Problema 20. Problema drapelelor Avem la dispoziție pânză de șapte culori: alb, galben, portocaliu, roșu, albastru, verde, negru. Se cere să se precizeze (eventual să se reprezinte grafic) toate drapelele tricolore care se pot confecționa, știind că trebuie respectate regulile:

- orice drapel are culoarea din mijloc alb, galben sau portocaliu;
- cele trei culori de pe drapel sunt distincte;
- drapelul poate fi împărțit în trei zone egale vertical sau orizontal.

R. Problema este o aplicație a generării aranjamentelor de șapte culori luate câte trei cu condiția suplimentară: culoarea din mijloc să fie alb, galben sau portocaliu.

Considerăm deci soluția un vector $x = (x_1, x_2, x_3)$, unde $x_i \in \{\text{alb, galben, portocaliu, roșu, albastru, verde, negru}\}$ ($\forall i=1,3$), mulțime pe care o codificăm $\{1,2,3,4,5,6,7\}$. Condițiile ca un drapel să fie acceptabil sunt $x_2 \in \{1,2,3\}$, $x_1 < x_2 < x_3$, iar pentru reprezentarea grafică observăm că unei soluții îi corespund două reprezentări: una verticală și una orizontală.

Algoritmul va fi următorul:

Procedure DrapeleTricolore (cul;x)

array cul(7),x(3)

k:=1

xk:=0

```

    while k>0:
    v:=0
    while xk<7:
    xk:=xk+1
    call Posibil (x,k,v)
    if v=1 then exit
    endif
    repeat
    if v=0 then k:=k-1
    else if k=3 then call AfișareVert(x,cul)
    call AfișareOriz(x,cul)
    else k:=k+1; xk:=0
    endif
    endif

```

repeat

return

end

Procedure Posibil(x,k,v)

array x(k) **if** k=2 **then if** xk $\in \{1,2,3\}$ **then** v:=0; **return**

endif for i=1.k-1

if xk=xi, **then** v:=0; **return**

endif repeat v:=1 return

Procedurile *AfişareVert* şi *AfişareOriz* reprezintă grafic drapelul cu zone egale vertical respectiv orizontal.

Problema 21. Rând de scaune Un grup de n persoane sunt aşezate pe un rând de scaune. Între oricare doi vecini izbucnesc conflicte de interese astfel încât aceştia se reaşază pe rând dar între oricare doi vechi vecini trebuie să existe una sau cel mult două persoane.

Să se realizeze un program care să afişeze toate variantele de reaşzare posibile.

R. Metoda de rezolvare este backtracking în varianta recursivă. Presupunem că inițial persoanele sunt numerotate în ordine de la stânga la dreapta cu $1, 2, \dots, n$.

Considerăm ca soluție un vector X cu n componente în care x_i are ca semnificație poziția pe care se va afla persoana i conform noii reaşzări. Pentru *lol* dat, condițiile de continuare sunt:

a) $x_j \neq x_k, (\forall) j=1, k-1$ (x trebuie să fie o permutare)

b) $|x_k - x_{k-1}| \in \{2, 3\}$ (între persoana k şi vecinul său anterior trebuie să se afle una sau două persoane).

În locul soluției X se va lista permutarea $Y = X^{-1}$ cu semnificația y_i este persoana care se aşază pe locul i .

Problema 22. Fotografia Se consideră fotografia alb-negru a unui obiect, reprezentată prin n puncte, pentru fiecare punct ştiind vecinii săi dați în matricea $V(n, 8)$.

Să se scrie un program pentru a testa dacă obiectul fotografiat este compus dintr-o singură bucată.

R. Fie matricea V matricea vecinilor unde $V(n, 8)$ şi 0 dacă punctul i nu are vecin pe poziția j ;

$V(i, j) = k$ dacă punctul i are vecin pe poziția j punctul k .

Folosim un algoritm recursiv de marcare pornind de la un punct a tuturor vecinilor săi direcți sau indirecti. Dacă după marcare mai există puncte nemarcate atunci figura nu este dintr-o bucată.

Fie m cu n componente, vectorul marcărilor:

$$m(i) = \begin{cases} 0 & \text{dacă } i \text{ nu este marcat;} \\ 1 & \text{dacă } i \text{ e marcat.} \end{cases}$$

Inițial m este tot zero. Pornim la marcare din punctul 1. Algoritmul recursiv este următorul:

Program **Fotografia**($n, v, \text{raspuns}$) array $v(n, 8), m(n)$ for $i=2, n$:

$m(i) := 0$ repeat $m(i) := 1$ call **Marcare**($1, v, m$) for $i=1, n$:

if $m(i) = 0$ then $\text{raspuns} := \text{"nu e dintr-o bucata"}$

stop endif repeat $\text{raspuns} := \text{"e dintr-o bucata"}$ **end**

Procedure **Marcare** (p, v, m)

array $v(n, 8), m(n)$

$m(p) := 1$

for $j=1, 8$

if $v(p, j) > 0 \wedge m(v(p, j)) = 0$ **then call** **Marcare**($v(p, j), v, m$) **endif**

repeat return end

Folosim deci pentru marcare procedura recursivă *Marcare()* care pornind de la un punct p marchează pe rând toți vecinii acestuia dacă există împreună cu vecinii vecinilor lor ş.a.m.d.

Observăm că înainte de a-l marca testăm dacă punctul are vecin pe poziția j şi dacă acesta nu este marcat. Putem înlocui condiția cu o singură verificare dacă mai introducem în m o componentă $m(0) = 1$ şi atunci vom avea:

if $m(v(p, j)) = 0$ **then call** **Marcare**($v(p, j), v, m$) **endif** verificându-se astfel implicit dacă există vecin pe poziția j .

Problema 23. Partițiile unei mulțimi. Fie A o mulțime finită. O partiție a lui A este o descompunere a sa de forma $A = A_1 \cup A_2 \cup \dots \cup A_k$ unde cele k submulțimii A_1, A_2, \dots, A_k numite clase sunt nevide şi două câte două disjuncte.

Determinați toate partițiile mulțimii A cu n elemente.

R. Folosim metoda backtracking în varianta recursivă. Utilizăm pentru reprezentarea partițiilor un vector v cu n componente. Elementul v_i va avea ca valoare numărul de ordine al clasei din care face parte elementul i . De exemplu dacă $v = (1, 3, 1, 4, 2, 2)$ atunci partiția va fi :

$A = \{a_1, a_3\} \cup \{a_5, a_6\} \cup \{a_2\} \cup \{a_4\}$ observând că în clase elementele sunt ordonate crescător după indice.

Procedura *Partitii* este o procedură recursivă, având parametrii k (cu semnificația că trebuie stabilit numărul clasei în care se va afla a) şi nc (desemnând numărul claselor partiției curente a mulțimii $\{a_1, a_2, \dots, a_{k-1}\}$).

Dacă valoarea lui k este $n+1$, înseamnă că vectorul v este construit în întregime şi ca urmare va fi apelată procedura de scriere. Dacă valoarea lui k nu depăşeşte pe n , atunci:

- pentru $i=1,2,\dots,k-1$ plasăm pe ak în clasa lui ai (adică atribuim lui vk valoarea vi și apelăm $Partitii(k+1,nc)$);
 - plasăm pe ak într-o nouă clasă (deci în clasa $nc+1$, ceea ce revine la a atribui lui vk valoarea $nc+1$) și apelăm $Partitii(k+1,nc+1)$.

```

Program PartitiiMultime;
uses Crt;
var v,a: array [1..100] of integer; n:integer;
    Procedure Scriere;
    var nec,j:integer;c,s:string;
    begin c:='A = ';
    for i:=1 to n do begin nec:=0;
    for j:=1 to n do if v[j]=i then nec:=nec+1;
    if nec>0 then begin c:=c+'{ ';
    for j:=1 to n do
    if v[j]=i then begin
    str(a[j],s);
    c:=c+s+', ' end;
    delete(c,length(c)-1,2);
    c:=c+' } U ' end;
    delete(c,length(c)-2,3);
    write(c); writeln;
    s:=ReadKey end;
    Procedure Partitii(k,nc:integer);
    var i:integer;
    begin
    if k=n+1 then Scriere else begin
    for i:=1 to nc do begin
    v[k]:=i;
    Partitii(k+1,nc) end;
    v[k]:=nc+1;
    Partitii(k+1,nc+1) end end;
    begin
    write('n= '); readln(n);
    for i:=1 to n do begin write('a[' ,i,']='); readln(a[i]) end;
    Partitii(1,0) end.
  
```

Problema 24. Rețeaua binară Se dau numerele naturale m și n . Se consideră o rețea dreptunghiulară având 2^m linii și 2^n coloane. Se cere să se atribuie nodurilor rețelei numere naturale distincte din mulțimea $\{0,1,\dots,2^{m+n}-1\}$, astfel încât pentru orice două vârfuri vecine (pe orizontală sau pe verticală) reprezentarea lor binară să difere pe o unică poziție.

R. Folosim metoda backtracking generând cele 2^{m+n} numere, verificând condițiile ca numărul din poziția $R(k-1)$ și $R(k-2^n)$ pentru k corespunzător să difere pe o unică poziție față de $R(k)$, $R(k) \in \{0,1,\dots,2^{m+n}-1\}$, $k=1,2^{m+n}$.

Algoritmul este următorul:

```

Procedure Posibil(k,R,v) array R( $2^{m+n}$ ) if  $k \bmod x^n \neq 1$  then call Verifica(k,k-1,v)
if  $v=0$  then return endif endif if  $k > 2^n$  then call Verifica(k,k- $2^n$ ,v)
return endif  $v:=1$  return; end

Procedure Verifica(t,l,v) array R( $2^{m+n}$ )  $x:=|R(t)-R(l)|$  while  $x \neq 1$  and  $x \neq 0$ 
if  $x/2 \equiv [x/2]$  then  $v:=0$ ; return
else  $x:=x/2$  endif repeat if  $x=0$  then  $v:=0$ 
else  $v:=1$  endif return end
  
```

```

Program Retea(m,n,R) array R( $2^{m+n}$ )  $k:=1$   $R_1:=1$  while  $k > 0$   $v:=0$  while  $R_k \leq 2^{m+n}-2$   $R_k:=R_k+1$  call
Posibil(k,R,v) if  $v=1$  then exit endif repeat if  $v=0$  then  $k:=k-1$ 
else if  $k=2^{m+n}$  then call Scrie(R,k)
stop else  $k:=k+1$ 
 $R_k:=1$  endif endif repeat write "Nu ex. sol. !!!" end
  
```

Problema 25. Problema descompunerii unui număr natural Orice număr natural se poate scrie ca sumă de termeni numere naturale diferite de zero. Astfel, numărul 4 se poate scrie: $4=3+1=2+2=2+1+1=1+1+1+1$, deci în patru moduri.

Dacă notăm cu n numărul pe care vrem să-l scriem ca sumă de numere naturale și cu $T(N)$ numărul de modalități în care se poate scrie n , atunci pentru $n=4$ avem $T(4)=4$.

Realizați un program care să calculeze și să afișeze pentru un n dat ($n \leq 50$) valoarea lui $T(N)$.

(Concursul de informatică, A.S.E., București, 1993)

R. O primă variantă de rezolvare este prin backtracking, soluțiile fiind prezentate sub forma unui vector (x_k) , $k=1,n$, unde n este numărul de descompus. Termenii rezultați din descompunere sunt memorați în elementele vectorului. Algoritmul are următoarele particularități:

- vectorii soluție au un număr variabil de componente (dar cel mult n), o condiție de revenire fiind ca suma elementelor să fie n ;

- prima valoare pe care poate s-o ia un element trebuie să fie cel puțin egală cu a elementului anterior, pentru a se evita numărarea unei soluții de două ori.

Metoda permite nu doar numărarea posibilităților de descompunere ci și afișarea soluțiilor găsite.

Programul scris în limbajul Pascal este:

```
Program DescompunereNumar;
type vect=array[1..100] of integer;
var x:vect; k,vb,n,s:integer; nrsum:longint;
    Function sum(var x:vect; k:integer):integer;
    var j,s:integer;
    begin
        s:=0;
        for j:=1 to k do s:=s+x[j];
        sum:=s end;
    begin
        k:=1; x[k]:=0; nrsum:=0;
        writeln('Introduceți numărul de descompus (0 pentru ieșire): ');
        repeat
            k:=1;x[k]:=0;nrsum:=0;
            write('N=');readln(n);
            while k>0 do begin vb:=0;
                if x[k]<n-1 then begin x[k]:=x[k]+1;
                    vb:=1
                end;
            end;
            if vb=1 then begin s:=sum(x,k);
                if s=n then
                    begin nrsum:=nrsum+1;
                        k:=k-1 end else if s>n then k:=k-1 else begin k:=k+1; x[k]:=x[k-1]-1 end end else k:=k-1 end;
            writeln('T('n:3,')='n:3,nrsum) until n=0 end.
```

O a doua variantă de rezolvare constă în folosirea unei funcții recursive de descompunere a numărului n . Numărul este descompus în câte doi divizori astfel:

$(n-k,k)$, $k=1, \lfloor n/2 \rfloor$, descompunerea urmând să se aplice apoi numărului $nu=n-k$ în felul următor: $(nu-i,i)$, $i=k, nu-k$, la fel pentru $nu-i$, ș.a.m.d.

Algoritmul are avantajul unei mult mai mari rapidități dar dezavantajul că nu poate prezenta variantele de descompunere.

Programul este următorul:

```
Program Termeni;
var nrsum:longint; n,sant,nl:integer;
    Procedure Divide(n,start:integer);
    Vardl,d2,rez:integer;
    begin
        d2:=start;
        d1:=n-d2;
        while d1>=d2 do begin nrsum:=nrsum+1;          rez:=d1-d2;if rez>=start then divide(d1,d2);
            d1:=d1-1; d2:=d2+1 end
        end;
    begin
        writeln('Introduceți numărul de descompus (Ieșire cu 0):');
        repeat
            write('N=');readln(n);
            nl:=n;
            sant:=1; nl:=n-sant; nrsum:=0;
            while nl>=sant do begin
                nrsum:=nrsum+1;
                divide(nl,sant);
                nl:=nl-1;sant:=sant+1 end;
```

writeln('T(',n:3,')=' ,nrsum) until n=0 end.

Problema 26. Subșiruri de lungime maximă Fie $x=(x_1,x_2,\dots,x_n)$ un vector cu componente distincte. Se cere să se determine toate subșirurile de lungime maximă :
 $(x(i_1), x(i_2), \dots, x(i_k))$ cu $i_1 < i_2 < \dots < i_k$ și $x(i_1) < x(i_2) < \dots < x(i_k)$.

R. Folosim metoda backtracking construind $S=(S_1, S_2, \dots, S_{\max})$ un subșir al lui x , unde $S_i \in \{x_{i_1}, x_{i_1+1}, \dots, x_{n-\max+i}\}$. Putem construi soluția ca un subșir de indici

$i=(i_1, i_2, \dots, i_{\max})$ unde $i_k \in \{k, k+1, \dots, n-\max+k\}$ și $i_k > i_{k-1}, \forall k > 1, x(i_k) > x(i_{k-1}), \forall k > 1$.

Pornim cu \max de la n și îl scădem apoi până găsim subșirurile. Algoritmul este următorul:

Program Subsir(x,n,i) **array** x(n,) **max**:=n; **t**:=0 **while** t=0
k:=1; **ik**:=k-1 **while** k>0 **v**:=0 **while** ik<n-max+k
ik:=ik+1; **call** Posibil(i,k,v) **if** v=1 **then exit endif repeat if** v=0 **then** k:=k-1
else if k=max **then call** Scribe(i,max)
t:=1 **else** k:=k+1
ik:=k-1 (sau i_{k-1}) **endif endif repeat** **max**:=max-1 **repeat end**

Procedure Posibil(i,k,v)

array i(k)
if k=1 **then** v:=1; **return**
endif * if $i_{k-1} \geq i_k$ **then** v:=0; **return**
endif
if $x(i_k) > x(i_{k-1})$ **then** v:=1 **else** v:=0
endif return; end

Condiția * poate fi eliminată dacă inițializăm i_k cu i_{k-1} ($i_k := i_{k-1}$).

Problema 27. Problema comis-voiajorului. Se dau n orașe, numerotate $1, 2, \dots, n$. Între orașe există legături directe, al căror cost se cunoaște. Un comis-voiajor vrea să plece dintr-un oraș, să se întoarcă în orașul de plecare, vizitând o singură dată fiecare oraș, pe un drum de cost total minim. Se cere să se furnizeze traseul comis-voiajorului.

Rezolvare. Considerând cele n orașe ca vârfuri ale unui graf neorientat, iar legăturile dintre orașe ca muchii ale grafului, problema se reduce la determinarea ciclului hamiltonian de cost total minim într-un graf neorientat (ciclul care trece prin toate vârfurile grafului și are costul total minim). Pentru această problemă nu se cunosc încă algoritmi polinomiali de rezolvare. Propunem două variante de rezolvare, cea de-a doua fiind o tehnică de Inteligență Artificială.

Specificarea problemei

Date

n - numărul de orașe

$a(n,n)$ - matricea reprezentând legăturile dintre orașe (matricea de adiacență a grafului), cu semnificația: $a[i,j]=c$, dacă între orașele i și j există o legătură directă al cărei cost este c , respectiv $a[i,j]=0$, dacă $i=j$ sau dacă între orașele i și j nu există legătură directă

Varianta 1. Algoritmul de rezolvare se bazează pe metoda *backtracking* și constă în a genera toate ciclurile hamiltoniene din graf și a-l reține pe cel de cost total minim.

Se vor genera toate succesiunile de orașe $x_1, x_2, \dots, x_n, x_{n+1}$ satisfăcând următoarele condiții:

- $x_1 = x_{n+1}$
- elementele vectorului sunt distincte (excepție făcând capetele care sunt egale)
- pentru orice $i=1, n$ x_i și x_{i+1} sunt adiacente (unite prin muchie)

Observații

Metoda folosită spre rezolvare

- garantează obținerea soluției optime
- nu este eficientă, datorită timpului ridicat de execuție, proporțional cu $n!$ (backtrackingul nu evită explozia combinatorială)

function posibil(a, n, k, x, c)

{verifică dacă elementul $x[k]$ este corect ales în soluție – c reprezintă costul traseului, dacă acesta este un ciclu hamiltonian}

if $k > n+1$ **then**
for $i \leftarrow 1$ **to** $k-1$ **do**
if $x[k] = x[i]$ **then return false**
endif
endfor
else
 $c = a[x[n], x[n+1]]$
if $x[k] < x[1]$ **then return false**

```

    endif
    for i ← 2 to k-1 do
        if x[k] = x[i] then return false
        endif
        c = c + a[x[i-1], x[i]]
    endfor

endif

if k > 1
    if a[x[k-1], x[k]] = 0 then return false
    endif

endif
return true;
{ ***** }
procedure compar(a, n, x, c, cmin, o)
{se verifică dacă costul ciclului hamiltonian x[1],...x[n+1] este minim; în caz afirmativ se reține ca fiind soluție a problemei}
    if c < cmin then
        cmin = c
        o = x

    endif
{ ***** }
procedure back(a, n, x, k, cmin, o)
{se generează recursiv toate soluțiile problemei și se reține soluția optimă}
    for i ← 1 to n do
        x[k] = i
        if posibil(a, n, k, x, c) then
            if k = n+1 then compar(a, n, x, c, cmin, o)
            else back(a, n, x, k+1, cmin, o)

        endif
    endfor
{ ***** }
{ programul principal – generează ciclul hamiltonian de cost total minim}
{ citirea datelor}
cmin = 1.0e6
{ se alege un oraș de plecare arbitrar}
x[1] = 1
{ se începe generarea soluției cu al doilea element}
back(a, n, x, 2, cmin, o)
if cmin = 1.0e6 then
    write "Nu exista soluție a problemei"
else
    { scrie soluția problemei: vectorul o[1],...o[n+1] reprezentând ordinea
    în care vor fi vizitate orașele – costul traseului este cmin }
endif
endif

```

Problema 28 (Divide et Impera): Se citește un vector cu n componente, numere naturale. Se cere să se tipărească valoarea maximă. Funcția căutată va genera valoarea maximă dintre numerele reținute în vector pe o poziție dintre i și j (inițial, i=1, j=n). Pentru aceasta, se procedează astfel:

- dacă i=j, valoarea maxima va fi v[i];
- în caz contrar, se împarte vectorul în doi subvectori - presupunem varianta pentru paralelizare pe 2 procesoare. Se calculează mijlocul m al intervalului [i, j]: $m = (i+j) \div 2$. Primul subvector va conține componentele de la i la m, al doilea va conține componentele de la (m+1) la j; se rezolvă subproblemele (aflându-se astfel maximum pentru fiecare din subvectori), iar soluția curentă va fi dată de valoarea maximă dintre rezultatele celor două subprobleme.

```

#include<iostream.h>
int v[10],n;
int max(int i, int j)
{ int a, b, m;

```

```

if (i==j) return v[i];
else
{
    m = (i+j)/2;
    a = max(i, m);
    b = max(m+1, j);
    if (a>b) return a;
    else return b;
}
}
main( )
{
    cout<<"n=";>>n;
    for (int i=1; i<=n; i++)
    {
        cout<<"v["<<i<<"]="; cin>>v[i];
    }
    cout<<"max="<<max(1,n);
}

```

Problema 29. Considerăm problema înmulțirii matricelor (*Programarea dinamica*):

Se consideră matricele A_1, A_2, \dots, A_n , cu dimensiunile specificate. Pentru a calcula produsul acestor matrice sunt necesare $n-1$ înmulțiri. Ordinea efectuării produselor nu este unic determinată, datorită proprietății de asociativitate. Dacă matricele au dimensiuni diferite, numărul operațiilor de înmulțire elementară pentru a obține matricea rezultat depinde de ordinea de înmulțire aleasă. Se cere să se găsească modalitatea optimă de asociere astfel încât numărul înmulțirilor să fie minim.

Rezolvare: Considerăm următorul exemplu:

$A \times B \times C \times D$, unde $A(13,5)$, $B(5,89)$, $C(89,3)$, $D(3,34)$. Modalitățile de asociere sunt:

| | |
|--------------|-----------------|
| $((AB)C)D$ | 10582 înmulțiri |
| $((AB)(CD))$ | 54201 înmulțiri |
| $((A(BC))D)$ | 2856 înmulțiri |
| $(A((BC)D))$ | 4055 înmulțiri |
| $(A(B(CD)))$ | 26418 înmulțiri |

Este clar acum că este esențial modul de asociere a înmulțirilor.

Modalitatea optimă de asociere se poate determina fără a calcula toate posibilitățile. Notăm cu $C_{i,j}$ numărul minim de înmulțiri elementare necesare calculării produsului A_i, A_{i+1}, \dots, A_j , pentru $1 \leq i \leq j \leq n$. Avem:

- $C_{i,i} = 0$
- $C_{i,i+1} = d_i \bullet d_{i+1} \bullet d_{i+2}$
- $C_{i,j}$ este valoarea minimă căutată
- este verificat principiul optimalității și
- $C_{i,j} = \min \{ C_{i,j} + C_{i,j} + d_i \bullet d_{k+1} \bullet d_{j+1} \mid 1 \leq k \leq j \}.$

Se va reține la fiecare pas indice care realizează minimul **MinNrMul**, adică modul de asociere al matricelor. Această valoare ne permite construirea arborelui binar care descrie ordinea efectuării operațiilor. Vârfurile arborelui vor conține limitele subșirului de matrice ce se asociază, rădăcina va conține $(1,n)$, iar un subarbore cu rădăcina (i,j) va avea ca descendent $[i, k]$ și $[k+1, j]$, unde k este valoarea pentru care se realizează optimul cerut. Parcurgerea arborelui în postordine indică ordinea efectuării produselor.

```

program OrdineInmultireMatrici;
const MaxMatrici = 100;
type PNod = ^TNod;
TNod = record
    ValSt, ValDr: Integer;
    ArbSt, ArbDr: PNod;
end;
var Nr: Integer; { Numărul matricelor }
Dim: array[1 .. MaxMatrici + 1] of Integer; { Dimensiunile matricelor }
NrMul: array[1 .. MaxMatrici + 1, 1 .. MaxMatrici + 1] of Integer;
{ Nr. înmulțirilor elementare necesare calculării prod.  $A_i \times \dots \times A_j$  }
{ Citirea datelor } procedure Citeste;
var I: Integer;
begin
    repeat Write('Numărul matricelor: '); Readln(Nr); until (Nr > 0) and (Nr <= MaxMatrici);
    Writeln('Șirul dimensiunilor:');
    for I := 1 to Nr + 1 do

```

```

    repeat Write('Dimensiunea ', I, ': '); Readln(Dim[I]); until Dim[I] > 0;
end;
{ Calculul numărului înmulțirilor }
function CalcNrMul(I, J, K: Integer): Integer; {  $I \leq K \leq J$  }
begin
    CalcNrMul := NrMul[I, K] + NrMul[K + 1, J] + Dim[I] * Dim[K + 1] * Dim[J + 1];
end;
{ Minimizarea numărului înmulțirilor }
function MinNrMul(I, J: Integer; var KMin: Integer): Integer;
var K: Integer;
begin
    KMin := I;
    for K := I + 1 to J - 1 do
        if CalcNrMul(I, J, K) < CalcNrMul(I, J, KMin) then KMin := K;
    MinNrMul := CalcNrMul(I, J, KMin);
end;
{ Construirea arborelui } function ConsArb(St, Dr: Integer): PNode;
var Arbore: PNode;
begin
    New(Arbore);
    with Arbore^ do
        begin
            ValSt := St; ValDr := Dr;
            if St < Dr then
                begin
                    ArbSt := ConsArb(St, NrMul[Dr, St]);
                    ArbDr := ConsArb(NrMul[Dr, St] + 1, Dr);
                end
            else
                begin
                    ArbSt := nil; ArbDr := nil;
                end;
            end;
            ConsArb := Arbore;
        end;
end;
{ Parcurgerea în postordine a arborelui } procedure PostOrdine(Arbore: PNode);
begin
    if Arbore <> nil then
        with Arbore^ do
            begin
                PostOrdine(ArbSt); PostOrdine(ArbDr); Write('(', ValSt, ',', ValDr, ');');
            end;
end;
{ Rezolvarea problemei } procedure Rezolva;
var Arbore: PNode; I, J, K, L: Integer;
begin
    for I := 1 to Nr do NrMul[I, I] := 0;
    for L := 1 to Nr - 1 do
        for I := 1 to Nr - L do
            begin
                J := I + L; NrMul[I, J] := MinNrMul(I, J, K);
                NrMul[J, I] := K; { Se memorează indicele minimului }
            end;
        end;
    Arbore := ConsArb(1, Nr); Writeln('Șirul asocierilor:'); PostOrdine(Arbore); Writeln;
end;
{ Programul principal } begin Citeste; Rezolva; end.

```

Exemplu: Elaborați un algoritm eficient care să afișeze parantezarea optimă a unui produs matricial $M(1), \dots, M(n)$. 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 "("
        paran(i, r[i, j])
        write "*"
        paran(r[i, j] + 1, j)
        write ")"

```

Arătați prin inducție că o parantezare completă a unei expresii de n elemente are exact $n-1$ perechi de paranteze. Deduceți de aici care este eficiența algoritmului.

Problema 30 (Metoda Branch and Bound). O problemă generală ce ar putea fi rezolvată utilizând această tehnică ar putea fi următoarea:

Se dă o configurație inițială, pe care o vom nota CI. Se știe că unei configurații i se pot aplica un număr finit de transformări (TR). Se cere șirul minim de transformări prin care se poate ajunge de la configurația inițială CI la o configurație finală CF dată.

Problema ar putea fi tratată folosind metoda **backtracking**, dar timpul de execuție va fi foarte mare, mai ales în cazul problemelor de dimensiuni mari. Având în vedere că nu se cere generarea tuturor soluțiilor

problemei, ci doar a soluției de lungime minimă, BB propune o variantă specifică de abordare, am putea spune că într-un anumit sens este o combinație a metodelor *backtracking* și *greedy*.

Rezolvarea problemei se bazează pe următoarea idee: s-ar putea construi un arbore, ale cărui noduri să conțină configurațiile ce pot fi obținute în urma aplicării tuturor transformărilor posibile. Vorbind la modul general, o parcurgere în lățime a arborelui (Bread-First), ar conduce la obținerea soluției de lungime minimă. Bineînțeles acest lucru practic nu este posibil, deoarece problema poate fi de dimensiune mare.

Prin *succesor* al unui nod în acest arbore vom înțelege o configurație în care se poate ajunge pornind de la nodul respectiv prin aplicarea uneia din transformările permise.

Având în vedere că în arbore anumite configurații se pot repeta și în plus dimensiunea arborelui (numărul de noduri) poate fi mare, nu se va reține întreg arborele, ci o listă cu nodurile ce trebuie prelucrate, pe care o vom numi în continuare **LIST**.

Prin *expandare* a unui nod se înțelege generarea tuturor succesorilor posibili ai nodului.

La un moment dat, un nod din **LIST** pot avea una din următoarele 2 stări: poate fi *expandat*, sau *neexpandat*, după cum a fost sau nu ales deja pentru expandare.

Problema este de a selecta pentru expandare un anumit nod din **LIST**, ținând cont de faptul că se va cere soluția de lungime minimă. Pentru acest lucru, se folosește un mecanism specific inteligenței artificiale, asociind fiecărui nod din listă o funcție de cost $f=g+h$, g reprezentând numărul de mutări necesare pentru a ajunge de la configurația inițială la configurația curentă, respectiv h reprezentând numărul (estimativ) de mutări necesare pentru a ajunge de la configurația curentă la configurația finală. Funcția h se numește *funcție euristică*, alegerea acesteia fiind importantă din punct de vedere al vitezei de execuție al algoritmului.

În plus, pentru a putea reface șirul mutărilor, va trebuie să reținem pentru fiecare nod din listă adresa "*părintelui*", adică a configurației din care a rezultat prin expandare configurația curentă.

În concluzie, nodurile din **LIST** sunt înregistrări ce conțin următoarele informații:

- configurația curentă
- valoarea funcției de cost f
- adresa nodului "*părinte*"
- adresa nodului următor din **LIST**
- un câmp care să indice dacă nodul a fost sau nu expandat

Algoritmul

1. Se va introduce în **LIST** un nod corespunzător configurației inițiale - pentru **CI** se va alege $g=0$ și $f=h$
2. Cât timp încă nu s-a ajuns la configurația finală și în **LIST** există noduri încă neexpandate, se execută:
 - se selectează din **LIST** nodul neexpandat având funcția de cost f minimă
 - se expandează acest nod, obținând toți succesorii acestuia
 - pentru fiecare succesor generat, se execută:
 - se calculează funcția g atașată succesorului, ca fiind valoarea lui g avută de părinte +1
 - se verifică dacă succesorul apare în lista **LIST** (a fost generat anterior)
 - în caz negativ, este trecut în **LIST** cu funcția de cost corespunzătoare ($f=g+h$) și este marcat ca fiind neexpandat
 - în caz afirmativ, se verifică dacă valoarea lui g actuală este mai mică decât valoarea lui g cu care a fost găsit în **LIST**
 - dacă da, nodul găsit în **LIST** este direcționat către actualul părinte și i se atașează noul g , iar dacă nodul a fost marcat ca fiind expandat, i se schimbă marcajul (devine neexpandat)
3. Dacă pasul 2 se încheie cu **LIST** vidă, înseamnă că problema nu are soluție.
4. Dacă pasul 2 se încheie cu selectarea configurație finale, se va reface șirul transformărilor de la configurația finală spre configurația inițială, utilizând legătura "*părinte*"

Observații

1. Datorită modului de generare al soluției (se selectează spre expandare nodul cel mai "promițător"), soluția obținută va fi de lungime minimă.
2. Se poate demonstra matematic acest lucru, folosind inducția matematică.

PROBLEMA31 "CĂSUȚE ADIACENTE" (Metoda Branch and Bound) Se consideră $2n$ căsuțe situate pe aceeași linie. Două căsuțe adiacente sunt goale, $n-1$ căsuțe conțin caracterul 'A', iar celelalte $n-1$ căsuțe conțin caracterul 'B'. O mutare constă în interschimbarea conținutului a două căsuțe adiacente nevide cu cele două căsuțe libere. La intrare este dată configurația inițială. Se cere să se determine un șir

minim de mutări prin care să se ajungă la configurația finală (cea în care toate caracterele 'A' apar înaintea caracterelor 'B')

Exemple

1. configurația inițială **ABB**BAA**
configurația finală **AAA**BBB**
șirul minim de mutări pentru a obține configurația finală
 - **ABB**BAA**
 - **ABBAAB****
 - **A**AABBB**
 - **AAA**BBB**
2. configurația inițială **AA**BBA**
configurația finală **AABAB****
șirul minim de mutări pentru a obține configurația finală
 - **AA**BBA**
 - **AABAB****
3. configurația inițială **BABABBA**ABABAB**
configurația finală **BBBBBBB**AAAAAA**
Nu există soluție
4. configurația inițială **ABB**AA**
configurația finală **ABAB**B**
Nu există soluție
5. configurația inițială **BAB**AA**
configurația finală ****BBAAA**
șirul minim de mutări pentru a obține configurația finală
 - **BAB**AA**
 - ****BBAAA**
 - **BB**AAA**
 - **BBAAA****
 - **B**AABA**
 - **BAA**BA**
 - ****ABABA**
 - **ABAB**A**

Observații

1. În configurații s-a înlocuit spațiul cu caracterul '*', pentru vizibilitate.
2. Configurațiile se vor reprezenta sub forma unor șiruri de caractere.
3. Pentru rezolvare se va aplica mecanismul BB. cu precizarea ca funcția euristică **h** se va alege astfel: dacă **C** este configurația curentă și **CF** este configurația finală, atunci **h(C)** reprezintă numărul de diferențe dintre configurațiile **C** și **CF**, adică numărul de caractere (diferite de spațiu) din **C** care nu se află pe poziția finală (din **CF**)
 - pentru exemplul 1
$$h('ABB**BAA')=1+1+1+1=4$$
$$h('A**AABBB')=1+1+1=3$$
 - pentru exemplul 2
$$h('AA**BBA')=1+1+1=3$$
4. În algoritmul ce-l vom prezenta, soluția va fi afișată în sens invers, sub forma șirului de mutări de la **CF** până la **CI**.
5. Dacă numărul de mutări necesare pentru a ajunge la configurația finală crește, timpul de execuție crește foarte mult (exponențial)

```
#include <stdio.h>
#include <conio.h>
#include <iostream.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#define maxint 32000
```

```

/* Structura unui nod
    - un câmp de informație inf - configurația curenta sub forma de string
    - g - funcția de cost asociata nodului
    - t - 0 daca nodul nu a fost expandat
      - 1 daca nodul a fost deja expandat
    - leg - adresa nodului următor din lista
    - sus - adresa nodului părinte (a configurației de unde a fost derivata
            configurația curenta) */

typedef struct nod
{
    char inf[100];
    int g,t;
    struct nod *sus,*leg;
} LISTA;
LISTA *cap;
/* funcția verifica daca o configurație x apare sau nu în lista – daca da, în pointerul q se retine adresa unde a fost găsită
configurația x în lista*/
int apare(char x[],LISTA *q)
{
    q=cap;
    while(q)
        if (!strcmp(q->inf,x)) return 1;
        else q=q->leg;
    return 0;
}
/* funcția determina numărul de apariții ale caracterului y în șirul de caractere x*/
int nr(char x[],char y)
{
    int k=0;
    for(int i=0;x[i];i++)
        k+=(x[i]==y)?1:0;
    return k;
}
/* funcția determina numărul de poziții pe care diferă configurația x de configurația finala */
int diferă(char x[])
{
    int k=0;
    for(int i=0;x[i];i++)
        k+=x[i]!=cf[i]?1:0;
    return k;
}
/* funcția interschimbă în x caracterele de pe pozițiile i,i+1 cu caracterele de pe pozițiile j,j+1. Rezultatul
interschimbării se retine în y */
void schimb(char x[],int i,int j,char y[])
{
    strcpy(y,x);
    char c=y[i];
    y[i]=y[j];
    y[j]=c;
    c=y[i+1];
    y[i+1]=y[j+1];
    y[j+1]=c;
}
/* eliberarea zonei de memorie ce a fost alocata listei */
void elib(void)
{
    while (cap)
    {
        LISTA *p=cap;
        cap=cap->leg;
        delete p;
    }
}

```

/* funcția care tipărește soluția problemei - y este configurația finala, configurațiile intermediare se vor obține pe baza legăturii SUS a nodurilor curente */

void tipar(LISTA *q,int găsit)

```
{
    LISTA *p;
    int k=1;
    if (!găsit) cout<<endl<<"nu este soluție";
    else
    {
        clrscr();
        cout<<endl<<k<<" "<<cf;
        int cond=1;
        while (cond)
            if (!strcmp(q->inf,ci)) cond=0;
            else
            {
                p=cap;
                int f=0;
                while (p&&!f)
                    if (!strcmp(p->inf,q->inf))
                    {
                        k++;
                        cout<<endl<<k<<" "<<p->inf;
                        f=1;
                    }
                    else p=p->leg;
                q=q->sus;
            }
        cout<<endl<<k+1<<" "<<q->inf;
        elib();
    }
    getch();
}
```

/* funcția care realizează expandarea unui nod q - daca în urma expandării s-a ajuns la configurația finala, y va conține configurația finala */

void expandare(LISTA *q,int &k,char y[])

```
{
    int j;
    LISTA *r,*p;
    char x[100];
    strcpy(x,q->inf);
    k++;
    j=strchr(x,' ')-x;
    strcpy(y,x);
    for(int i=0;i<=strlen(x)-2;i++)
        // if ((i!=j-1)&&(i!=j)&&(i!=j+1)&&(i!=j+2))
        if (!isspace(x[i])&&!isspace(x[i+1]))
        {
            schimb(x,i,j,y);
            if (!strcmp(y,cf))
                break;
            else
            {
                int v=k+difera(y);
                if (!apare(y,p))
                {
                    r=new LISTA;
                    strcpy(r->inf,y);
                    r->sus=q;
                    r->leg=cap;
                    r->t=0;
                    cap=r;
                }
            }
        }
}
```

```

        r->g=v;
    }
    else
        if (p->g>v)
        {
            p->g=v;
            p->sus=q;
            if (p->t)
                p->t=!p->t;
        }
    }
}

/* se realizează expandarea repetata a nodurilor din lista - pentru expandare se va alege nodul pentru care funcția de
cost este minima*/
void b_b(void)
{
    LISTA *q;
    char y[100];
    int găsit;
    cap=new LISTA;
    strcpy(cap->inf,ci);
    cap->leg=NULL;
    cap->sus=NULL;
    cap->t=0;
    cap->g=diferă(ci);
    int cond=1;
    int k=0;
    while (cond)
    {
        găsit=0;          // găsit=0 daca nu mai exista noduri neexpandate
        LISTA *p=cap;
        int min=maxint;
        while (p)
        {
            min=(!p->t && (min>p->g))?(q=p,p->g):min;
            p=p->leg;
        }
        if (min==maxint) cond=0;
        else
        {
            găsit=1;
            q->t=! q->t;
            expandare(q,k,y);
            if (!strcmp(y,cf)) cond=0;
        }
    }
    tipar(q,găsit);
}

void main(void)
{ char ci[20],cf[20];
  clrscr();
  cout<<"Dați configurația inițială:";
  cin>>ci;
  cout<<"Dați configurația finală:";
  cin>>cf;
  if ((nr(ci,'a')!=nr(cf,'a'))||(nr(ci,'b')!=nr(cf,'b')))
  {
      cout<<endl<<"nu este soluție";
      getch();
  }
  else
      b_b();
}

```

}

PROBLEMA 32 Mergesort (sortarea prin interclasare, Divide et Impera) Fie $T[1 .. n]$ un tablou pe care dorim sa-l sortam crescător. Prin tehnica divide et impera putem proceda astfel: separam tabloul T în doua părți de mărimi cat mai apropiate, sortam aceste părți prin apeluri recursive, apoi interclasam soluțiile pentru fiecare parte, fiind atenți sa 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  $U[1 .. n \text{ div } 2]$ ,  $V[1 .. (n+1) \text{ div } 2]$ 
     $U \leftarrow T[1 .. n \text{ div } 2]$ 
     $V \leftarrow T[1 + (n \text{ div } 2) .. n]$ 
    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 doua tablouri deja sortate U și V .

33. Determinarea maximului unui sir

O problema simpla, care poate fi rezolvata extrem de ușor, dar pe care încercam sa o abordam folosind DIVIDE ET IMPERA, pentru a ilustra tehnica,

Problema Sa se calculeze maximul șirului de numere întregi x_1, \dots, x_n .

Folosim o funcție recursiva MAXIM(s, d) care calculează maximul șirului cuprins între indicii s și d , folosind tehnica DIVIDE ET IMPERA.

Ideea algoritmului DIVIDE ET IMPERA este următoarea:

- pentru un anumit sir cuprins între indicii s și d (pornim inițial cu întregul sir) verificam daca are sau nu un singur element
- daca are un singur element, acesta va fi chiar elementul maxim
- daca șirul nu are un singur element, se împarte în 2 subșiruri: primul cuprins între indicii s și $(s+d) \text{ div } 2$, respectiv al doilea cuprins între indicii $(s+d) \text{ div } 2 + 1$ și d . Folosind funcția recursiva MAXIM (aici apare DIVIDE ET IMPERA) se calculează maximul m_1 , respectiv m_2 din cele doua subșiruri, maximul șirului fiind de fapt cel mai mare dintre m_1 și m_2

```

#include<stdio.h>
#include<conio.h>
#include<iostream.h>
int n,x[20];
/*****/
void citire(void)
{ int i;
  FILE *pf=fopen("fis.txt","r"); fscanf(pf,"%d",&n);
  for(i=1;i<=n;i++)   fscanf(pf,"%d",&x[i]); fclose(pf);  }
/*****/
int maxim(int s,int d)
{ int m,m1,m2;
  if (s==d) return x[s]; else { m=(s+d)/2; m1=maxim(s,m); m2=maxim(m+1,d); return
  m1>m2?m1:m2; }}
/*****/
void main(void)
{ citire(); clrscr(); cout<<"Maximul șirului este:"<<maxim(1,n); getch(); }

```

34..Problema “gropi și pomi”

O livada este specificata sub forma unui tablou pătratic de dimensiune $n \times n$, formata din pătratele de latura egala cu unitatea, în care se găsesc gropi sau pomi. Se cere sa se determine dreptunghiul de arie maxima ce are în interior doar pomi.

Algoritmul folosit este DIVIDE ET IMPERA:

Livada se va reprezenta sub forma matricei $A(n \times n)$, un element al matricei fiind 0 daca pe poziția respectiva este groapa, respectiv 1 daca pe poziția respectiva este pom. Vom nota cu (x_2, y_2) , respectiv (x_1, y_1) coordonatele coltului stânga-sus, respectiv dreapta-jos ale dreptunghiului de arie maxima ce conține în interior doar pomi.

Ideea algoritmului este următoarea:

- pentru un anumit dreptunghi (pornim inițial cu întreaga matrice) verificăm dacă avem sau nu o groapă în el (se va căuta de fapt prima groapă)
- dacă avem o astfel de groapă, problema se va descompune în patru subprobleme, având în vedere că de pe poziția gropii putem face o tăietură verticală sau orizontală.
- dacă în dreptunghi nu există nici o groapă, vom compara aria lui cu aria maximă, dacă este cazul îl vom reține ca fiind dreptunghiul de arie maximă.

```
#include <stdio.h>
#include <conio.h>
#include <iostream.h>
int n,a[20][20],x1=0,x2=0,y1=0,y2=0;
//*****

void citire(void)
{ FILE *pf=fopen("fis.txt","r"); fscanf(pf,"%d",&n);
  for(int i=1;i<=n;i++)
    for(int j=1;j<=n;j++)    fscanf(pf,"%d",&a[i][j]);    fclose(pf);    }
//*****

void divimp(int xs,int ys,int xj,int yj,int& x2,int& y2,int& x1,int&y1)
{ int găsit=0,i1,j1;
  for(int i=xs;i<=xj&&!gasit;i++)
    for(int j=ys;j<=yj&&!gasit;j++)
      if (a[i][j]) { i1=i; j1=j; găsit=1; }
  if (!găsit)
    if ((xj-xs+1)*(yj-ys+1)>(x1-x2+1)*(y1-y2+1)) { x2=xs; y2=ys; x1=xj; y1=yj; } else;
    else { divimp(xs,ys,xj,j1-1,x2,y2,x1,y1);    divimp(xs,j1+1,xj,yj,x2,y2,x1,y1);
      divimp(xs,ys,i1-1,yj,x2,y2,x1,y1);    divimp(i1+1,ys,xj,yj,x2,y2,x1,y1);    }    }
//*****

void main(void)
{ citire(); clrscr(); divimp(1,1,n,n,x2,y2,x1,y1);    cout<<"\n"<<"Aria maxima este :"<<(x1-x2+1)*(y1-
y2+1);
  cout<<" iar dreptunghiul este :";    cout<<endl<<(" "<<x2<<" "<<y2<<" "<<"-----")<<(" "<<x1<<" "<<y1<<"");
  getch(); }
```

35. Problema “plierii”

Se considera un vector de lungime n . Definim plierea vectorului prin suprapunerea unei jumătăți numită donatoare peste cealaltă jumătate numită receptoare, cu precizarea că dacă numărul de elemente este impar, elementul din mijloc este eliminat. În acest mod se ajunge la un subvector ale cărui elemente au numerotarea corespunzătoare jumătății receptoare.

Plierea se poate aplica repetat până când se ajunge la un subvector de lungime 1 numit element final. Se cere să se precizeze toate elementele finale.

Algoritmul folosit este DIVIDE ET IMPERA:

Vom utiliza un vector f_1, \dots, f_n având valori binare 0 și 1, cu semnificația: f_i este 1 dacă elementul al i -lea din sir se poate obține ca rezultat al unei plieri, respectiv 0 în caz contrar. Folosim procedura recursivă PLIERE(s, d) care efectuează operația de pliere pentru subșirul cuprins între indicii s și d . (procedura PLIERE ilustrează de fapt tehnica DIVIDE ET IMPERA)

Ideea algoritmului DIVIDE ET IMPERA este următoarea:

- pentru un anumit subșir cuprins între indicii s și d (pornim inițial cu întregul sir) verificăm dacă are sau nu un singur element
- dacă are un singur element, acesta este element final
- dacă subșirul nu are un singur element, apar două posibilități:
 - dacă șirul are un număr impar de elemente, elementul din mijloc nu poate fi element final, el va fi eliminat, după care plierea sa se reduce la plierea a 2 subșiruri: subșirul cuprins între indicii s și $(s+d) \div 2 - 1$, respectiv cel cuprins între indicii $(s+d) \div 2 + 1$ și d (aici apare tehnica DIVIDE ET IMPERA)
 - dacă șirul are un număr par de elemente, nu există element de mijloc care să fie eliminat, de aceea plierea sa se reduce la plierea a 2 subșiruri: subșirul cuprins între indicii s și $(s+d) \div 2$, respectiv cel cuprins între indicii $(s+d) \div 2 + 1$ și d

```

#include<stdio.h>
#include<conio.h>
#include<iostream.h>
int *f,n;
void pliere(int s,int d)
{ int m=(s+d)/2;
  if (s==d) f[d]=1;   else  if (!(s+d)%2)) { f[m]=0; pliere(s,m-1);   pliere(m+1,d); }
  else { pliere(s,m);   pliere(m+1,d);      } }
/*****/
void scrie(void)
{ cout<<"Elementele finale sunt :\"endl;
  for(int i=1;i<=n;i++)   if (f[i]) cout<<i<<\" \"; getch(); }
/*****/
void main(void)
{ clrscr(); cout<<\"n=\"; cin>>n; f=new int[n+1]; pliere(1,n); scrie(); delete f; }

```

36. Problema “triunghiului”

Se considera un triunghi de numere naturale format din n linii în care prima linie conține un număr, a doua linie două numere, ..., a n -a linie n numere. Se cere să se calculeze cea mai mare sumă ce se poate forma cu numerele situate pe drumurile ce pleacă de pe prima linie și ajung pe ultima linie, astfel încât succesorul unui număr se afla pe linia următoare plasat sub el (aceeași coloană) sau pe diagonală la dreapta (coloana crește cu 1).

Datele se vor memora sub forma unei matrice triunghiulare inferioare $A(n,n)$.

O prima metoda de rezolvare, dar care nu este deloc eficientă, ar fi **backtracking**, să generăm toate sumele ce se pot forma (care sunt în număr de 2^{n-1}) și dintre toate acestea să reținem suma maximă,

La o analiză mai atentă, problema se poate rezolva folosind PD. Descrierea algoritmului de PD pentru rezolvarea acestei probleme presupune următoarele precizări:

Structura soluției optime

Se va construi o matrice triunghiulară inferioară m având următoarea semnificație: $m[i,j]$ = suma maximă ce se poate obține pe un drum de pe poziția (1,1) a matricei și până pe poziția (i,j). Se verifică principiul optimalității astfel: dacă suma elementelor de pe linia 1 până pe linia n este maximă (optimumul general), atunci și suma elementelor de pe linia 1 până pe linia i este maximă (optimumul parțial) - în caz contrar s-ar contrazice ipoteza.

Definierea recursivă a valorii soluției optime

- $m[1,1] = a[1,1]$
- pentru $i=2,n$

$$m[i,1] = m[i-1,1] + a[i,1]$$

$$m[i,i] = m[i-1,i-1] + a[i,i]$$

pentru $j=2,i-1$

$$m[i,j] = \max \{m[i-1,j-1] + a[i,j], m[i-1,j] + a[i,j]\}$$
- Suma maximă cerută va fi cea mai mare valoare de pe ultima linie a matricei m , adică
$$s_{\max} = \max \{m[n,1], m[n,2], \dots, m[n,n]\}$$

Observație

- ordinul de complexitate al algoritmului este $O(n^2)$
- Dacă vrem să determinăm și elementele din matrice ce compun suma maximă, vom reține o matrice suplimentară **poz** cu semnificația $poz[i,j]$ = coloana în care se găsește predecesorul lui $m[i,j]$.

```

#include <stdio.h>
#include <conio.h>
#include <iostream.h>
int n,a[20][20],m[20][20];
/*****/
void citire(void)
{ FILE *pf=fopen("mat.txt","r"); fscanf(pf,"%d",&n);
  for(int i=1;i<=n;i++)
    for (int j=1;j<=i;j++) fscanf(pf,"%d",&m[i][j]); fclose(pf);}
/*****/

```

```

void triunghi(void)
{ int i;    m[1][1]=a[1][1];
  for(i=2;i<=n;i++) { m[i][1]=m[i-1][1]+a[i][1]; m[i][i]=m[i-1][i-1]+a[i][i]; }
  for(i=2;i<=n;i++)
    for(int j=2;j<=i-1;j++) m[i][j]=m[i-1][j-1]+a[i][j]>m[i-1][j]+a[i][j]?m[i-1][j-1]+a[i][j]:m[i-1][j]+a[i][j];
  int smax=0;
  for(i=1;i<=n;i++) smax=smax<m[n][i]?m[n][i]:smax; clrscr(); cout<<"Suma maxima este: "<<smax;
  getch();}
//*****
void main(void)
{ citire(); triunghi(); clrscr(); getch(); }

```

37 Subșir crescător de lungime maxima

Se considera șirul a_1, \dots, a_n . Sa se tipărească subșirul crescător de lungime maxima.

Structura soluției optime. Se va construi un vector $l[i]$, $i=1, n$ cu următoarea semnificație: $l[i]$ = lungimea maxima a subșirului crescător care are ca ultim element pe $a[i]$. În acest caz se verifica principiul optimalității astfel: daca $a_{i1}, a_{i2}, \dots, a_k, \dots, a_{im}$ este lungimea maxima a subșirului crescător care conține elementul de pe poziția k al șirului inițial, atunci subșirul $a_{i1}, a_{i2}, \dots, a_k$ este cel mai lung subșir crescător ce se termina cu elementul a_k (în caz contrar s-ar contrazice ipoteza)

Definirea recursivă a valorii soluției optime

- $l[1] = 1$
 - pentru $i=2, n$

$$l[i] = \max \{1 + l[j] \mid a[j] < a[i], j=1, 2, \dots, i-1\}$$
 - Lungimea maxima a subșirului crescător va fi data de cea mai mare valoare din șirul l_1, l_2, \dots, l_n .
- Pentru a determina elementele ce fac parte din subșirul crescător de lungime maxima vom retine un șir p_1, \dots, p_n c semnificația: p_i = indicele elementului ce precede pe a_i în subșirul crescător de lungime maxima ce se termina cu elementul de pe poziția i . (daca nu mai sunt elemente în stânga lui a_i în soluția optima, atunci $p_i=0$)

Observație

- subșirul crescător de lungime maxima se retine în șirul *subsir₁, subsir₂, ..., subsir_{lg}*.
- ordinul de complexitate al algoritmului este $O(n^2)$

```

#include <stdio.h>
#include <conio.h>
#include <iostream.h>
int n,a[20],x[20],sol=0;
//*****
void citire(void)
{ FILE *pf=fopen("sir.txt","r"); fscanf(pf,"%d",&n);
  for(int i=1;i<=n;i++) fscanf(pf,"%d",&a[i]); fclose(pf); }
//*****
void crescator(void)
{ int i,j,l[20],p[20],subsir[20],max,poz;
  l[1]=1; p[1]=0;
  for (i=2;i<=n;i++) { p[i]=0;max=1;
    for(j=1;j<=i-1;j++) max=(a[i]>a[j]) && (max<l[j]+1)?p[i]=j,l[j]+1:max; l[i]=max; }
  max=1; for(i=1;i<=n;i++) max=max<l[i]?poz=i,l[i]:max; clrscr(); int lg=0;
  while (p[poz]) { subsir[++lg]=a[poz]; poz=p[poz]; }
  subsir[++lg]=a[poz]; printf("\nSubsirul crescator de lungime maxima este\n");
  for(i=lg;i>=1;i--) cout<<subsir[i]<<" "; getch(); }
//*****
void main(void)
{ citire(); crescator(); }

```

38 Problema “sumelor”

Se considera șirul a_1, \dots, a_n . Sa se determine toate sumele ce se pot obține cu elemente distincte ale șirului.

Structura soluției

Se va construi un vector $c[i]$, $i=1, \text{smax}$ (smax fiind suma maxima ce se poate obține cu elementele șirului) cu următoarea semnificație: $c[i] = 1$ dacă i se poate obține ca suma de elemente distincte din șir, 0 în caz contrar. În acest caz se verifica principiul optimalității astfel: dacă $c[S] = 1$ și suma S conține elementul a_i , atunci și $c[S - a_i] = 1$.

Definirea recursivă a valorii soluției

- $c[0] = 1$; $\text{smax} = 0$
- pentru $i=1, n$
 pentru $j=\text{smax}, 0$ pas -1
 dacă $c[j]=1$ atunci $c[j+a[i]]=1$

Observație

- algoritmul se poate adapta astfel încât să se retina eventual și elementele ce compun sumele
- ordinul de complexitate al algoritmului este $O(n^2)$
- aceasta abordare este o variantă a **Algoritmului lui LEE**

```
#include <stdio.h>
#include <conio.h>
#include <iostream.h>
int n,a[20],c[10000],smax;
//*****
void citire(void)
{ FILE *pf=fopen("lee.txt","r"); fscanf(pf,"%d",&n);
  for(int i=1;i<=n;i++) fscanf(pf,"%d",&a[i]); fclose(pf); }
//*****
void scrie(void)
{ clrscr(); cout<<"Sumele ce se pot obține cu elementele șirului sunt:"<<"\n";
  for(int i=0;i<=smax;i++) if (c[i]) cout<<"\n"<<"Se poate obține suma : "<<i; getch(); }
//*****
void sume(void)
{ c[0]=1; smax=0;
  for(int i=1;i<=n;i++) { for(int j=smax;j>=0;j--) c[j+a[i]]=c[j]?1:c[j+a[i]]; smax+=a[i]; } }
//*****
void main(void)
{ clrscr(); citire(); sume(); scrie(); }
```

39 Problema “desertului”

O regiune deșertică este reprezentată printr-un tablou de dimensiune $m \times n$. Elementele tabloului sunt numere naturale reprezentând diferențele de nivel fata de nivelul mării (cota 0). Se cere să se stabilească un traseu pentru a traversa deșertul de la Nord la Sud (de la linia 1 la linia m) astfel;

- se pornește dintr-un punct al liniei 1
- deplasarea se poate face pe una din direcțiile SE,S,SV,E
- suma diferențelor de nivel (la urcare }I coborâre) să fie minimă

Presupunem ca matricea inițială este $H(m,n)$, $h_{i,j}$ reprezentând diferența de nivel a poziției de coordonate (i,j) fata de nivelul mării.

Structura soluției optime

Se va construi o matrice \underline{dif} având următoarea semnificație: $\underline{dif}[i,j]$ = suma minimă a diferențelor de nivel (la urcare }I coborâre) ce se poate obține pe un drum de pe poziția $(1,1)$ a matricei și până pe poziția (i,j) - cu respectarea condițiilor impuse de problema. Se verifica principiul optimalității astfel: dacă traseul optim de pe linia 1 până pe linia n trece prin elementul (i,j) , atunci și traseul de pe linia 1 până pe linia i este optim, în caz contrar s-ar contrazice ipoteza.

Definirea recursivă a valorii soluției optime

Problema este puțin mai complicată, deoarece se permit și deplasările spre E (pe aceeași linie). Presupunând ca am construit primele $i-1$ linii ale matricei \underline{dif} , construim linia i luând în considerare pentru început doar deplasările spre SE,S,SV. Apoi, actualizăm linia de n ori astfel: pentru fiecare element de pe linia i verificăm dacă este mai convenabilă o deplasare spre E; dacă da, actualizăm elementul corespunzător.

- $\underline{dif}[1,j] = 0 \quad \forall j = 1, 2, \dots, n$
- pentru $i=2, n$

- pentru fiecare $j=1,2,\dots,n$ luam în considerare deplasările dinspre NV (din punctul $(i-1,j-1)$), dinspre N (din punctul $(i-1,j)$) și dinspre NE (din punctul $(i-1,j+1)$)

$$dif[i,j] = \max \left\{ \begin{array}{l} dif[i-1,j-1] + abs(h[i,j] - h[i-1,j-1]), \\ dif[i-1,j] + abs(h[i,j] - h[i-1,j]), \\ dif[i-1,j+1] + abs(h[i,j] - h[i-1,j+1]) \end{array} \right\}$$

- de n ori se executa
 - pentru fiecare $j=1,2,\dots,n$ se verifica daca deplasarea dinspre V (din punctul $(i,j-1)$) nu este mai convenabila

$$dif[i,j] = \max \left\{ dif[i,j], \right. \\ \left. dif[i,j-1] + abs(h[i,j] - h[i,j-1]) \right\}$$

- Suma minima ceruta va fi cea mai mica valoare de pe ultima linie a matricei **dif**, adică

$$smin = \min\{dif[m,1], dif[m,2], \dots, dif[m,n]\}$$

Pentru a putea determina pozițiile prin care trece traseul optim, reținem o matrice suplimentară **c(m,n)**, unde $c_{ij} \in \{1,2,3,4\}$, cu semnificația: 1 - dacă deplasarea s-a făcut dinspre V (din punctul $(i-1,j)$), 2 - dacă deplasarea s-a făcut dinspre NV (din punctul $(i-1,j-1)$), 3 - dacă deplasarea s-a făcut dinspre N (din punctul $(i-1,j)$) și 4 - dacă deplasarea s-a făcut dinspre NE (din punctul $(i-1,j+1)$),

Observație

- ordinul de complexitate al algoritmului este $O(n^3)$
- determinarea traseului se va face recursiv, prin procedura **DRUM(i,j)**

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
int n,m,h[100][100],dif[100][100],c[100][100];
/*****/
void citire(void)
{ int i,j;
  FILE *pf; pf=fopen("desert.txt","r"); fscanf(pf,"%d %d",&m,&n);
  for(i=1;i<=m;i++) for(j=1;j<=n;j++) fscanf(pf,"%d",&h[i][j]); fclose(pf); }
/*****/
void calcul(void)
{ int i,j,l;
  for(j=1;j<=n;j++) dif[1][j]=0;
  for(i=1;i<=m;i++) { dif[i][0]=dif[i][n+1]=10000; h[i][0]=h[i][n+1]=0; }
  for (i=2;i<=m;i++) {
    for(j=1;j<=n;j++) { dif[i][j]=dif[i-1][j-1]+abs(h[i][j]-h[i-1][j-1]); c[i][j]=2;
      if (dif[i][j]>dif[i-1][j]+abs(h[i][j]-h[i-1][j])) { dif[i][j]=dif[i-1][j]+abs(h[i][j]-h[i-1][j]); c[i][j]=3; }
      if (dif[i][j]>dif[i-1][j+1]+abs(h[i][j]-h[i-1][j+1])) { dif[i][j]=dif[i-1][j+1]+abs(h[i][j]-h[i-1][j+1]);
        c[i][j]=4; } }
    for(l=1;l<=n;l++)
      for(j=1;j<=n;j++)
        if (dif[i][j]>dif[i][j-1]+abs(h[i][j]-h[i][j-1])) { dif[i][j]=dif[i][j-1]+abs(h[i][j]-h[i][j-1]); c[i][j]=1; } }
  }
/*****/
void drum(int i,int j)
{ switch (c[i][j]) {
  case 1:drum(i,j-1);break;
  case 2:drum(i-1,j-1);break;
  case 3:drum(i-1,j);break;
  case 4:drum(i-1,j+1);break; }
  printf("(%d,%d)",i,j); if (i!=m) printf("-"); }
/*****/
void minim(void)
{ int j,min,poz; min=32000;
```

```

for(j=1;j<=n;j++) min=(dif[m][j]<min)?(poz=j,dif[m][j]):min;
clrscr(); printf("Suma diferențelor de nivel este %d\n",min); printf("Traseul este:\n"); drum(m,poz);
getch();}
/*****/
void main(void)
{ citire(); calcul(); minim();}

```

40. Problema “vaselor cu apa”

Vasele 1, 2, 3 au volumele A_1, A_2, A_3 ($A_1, A_2, A_3 \leq 30$ litri). Inițial vasul 1 este plin cu apa. Se poate face o singură operație și anume se toarnă apa dintr-un vas până când acesta se golește sau până când vasul în care se toarnă se umple.

Problema este ca dintr-un număr minim de asemenea operații să se ajungă la situația ca într-unul din vase să se afle o cantitate Q de apa.

Structura soluției optime

Vom utiliza un tablou tridimensional **cant[a1,a2,a3]** cu semnificația: $\text{cant}[i,j,k]$ = numărul minim de operații pentru a ajunge din situația inițială în situația în care în vasul 1 se afla i unități de apa, în vasul 2 se afla j unități de apa și în vasul 3 se afla k unități de apa.

Soluția algoritmului este data de

$\min\{\text{cant}[Q,j_1,k_1], \text{cant}[i_1,Q,k_2], \text{cant}[i_2,j_2,Q]\} \forall i_1, i_2, j_1, j_2, k_1, k_2$

Vom nota în continuare cu $S[i,j,k]$ starea în care vasul 1 se afla i unități de apa, în vasul 2 se afla j unități de apa și în vasul 3 se afla k unități de apa.

Definirea recursivă a valorii soluției optime

iii. se inițializează tabloul tridimensional **cant** în felul următor:

$\text{cant}[a1,0,0]=0$ - corespunzător stării inițiale $S[a1,0,0]$

$\text{cant}[i,j,k]=100$ - $i \neq a1, \forall j=1, \dots, a2 \} \forall k=1, \dots, a3$ (stările

$S[i,j,k]$ încă nu au fost generate

iv. Folosim procedura recursivă CALCUL(i,j,k) care analizează toate posibilitățile de modificare a stării $S[i,j,k]$

- vasul 1 se toarnă în vasul 2 sau 3
- vasul 2 se toarnă în vasul 1 sau 3
- vasul 3 se toarnă în vasul 1 sau 2

Vom analiza situațiile ce apar doar pentru una din situațiile de mai sus (celelalte se tratează analog).

Sa presupunem ca în starea $S[i,j,k]$ vasul 1 se poate turna în vasul 2, adică $i \neq 0$ și $j < a2$. Apar două situații

- $j+i \leq a2$, caz în care vasul 1 se poate turna în întregime în vasul 2. În urma efectuării acestei operații rezulta starea $S[0,i+j,k]$
 - dacă numărul de operații prin care s-a obținut starea $S[0,i+j,k]$ este minim
 - se actualizează valoarea $\text{cant}[0,i+j,k]$ astfel: $\text{cant}[0,i+j,k] := 1 + \text{cant}[i,j,k]$ (starea $S[0,i+j,k]$ s-a obținut din starea $S[i,j,k]$ prin aplicarea unei operații suplimentare)
 - se apelează recursiv CALCUL($0,i+j,k$) pentru noua stare obținută,
- $j+i > a2$, caz în care vasul 1 se poate turna parțial în vasul 2. În urma efectuării acestei operații rezulta starea $S[i-a2+j,a2,k]$
 - dacă numărul de operații prin care s-a obținut starea $S[i-a2+j,a2,k]$ este minim
 - se actualizează valoarea $\text{cant}[i-a2+j,a2,k]$ astfel: $\text{cant}[i-a2+j,a2,k] := 1 + \text{cant}[i,j,k]$ (starea $S[i-a2+j,a2,k]$ s-a obținut din starea $S[i,j,k]$ prin aplicarea unei operații suplimentare)
 - se apelează recursiv CALCUL($i-a2+j,a2,k$) pentru noua stare obținută

Observații

- se poate adapta algoritmul astfel încât să se țină pe lângă numărul minim și operațiile efectuate
- în variabila **minim** se va ține numărul minim de operații cerut de problema

```

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
int a1,a2,a3,cant[30][30][30],q,minim;

```

```

/*****/
void citire(void)
{ FILE *pf; pf=fopen("apa.txt","r"); fscanf(pf,"%d %d %d %d",&a1,&a2,&a3,&q); fclose(pf); }
/*****/
void calcul(int i,int j,int k)
{ if ((i==q) || (j==q) || (k==q))
    minim=minim>cant[i][j][k]?cant[i][j][k]:minim; if (i&&(j<a2)) if (j+i<=a2)
/* se toarnă în întregime conținutul vasului 1 în vasul 2 */
    if (1+cant[i][j][k]<cant[0][j+i][k]) { cant[0][j+i][k]=1+cant[i][j][k]; calcul(0,j+i,k); }
    else; else /* se toarnă din vasul 1 în vasul 2 pana acesta se umple */
    if (1+cant[i][j][k]<cant[i-a2+j][a2][k]) { cant[i-a2+j][a2][k]=1+cant[i][j][k]; calcul(i-a2+j,a2,k);
}
if (i&&(k<a3)) /* se toarnă în întregime conținutul vasului 1 în vasul 3 */
    if (k+i<=a3)
        if (1+cant[i][j][k]<cant[0][j][k+i]) { cant[0][j][k+i]=1+cant[i][j][k]; calcul(0,j,k+i); }
        else; else /* se toarnă din vasul 1 în vasul 3 pana acesta se umple */
        if (1+cant[i][j][k]<cant[i-a3+k][j][a3])
            { cant[i-a3+k][j][a3]=1+cant[i][j][k]; calcul(i-a3+k,j,a3); }
if (j&&(i<a1)) /* se toarnă în întregime conținutul vasului 2 în vasul 1 */
    if (i+j<=a1) if (1+cant[i][j][k]<cant[j+i][0][k])
        { cant[j+i][0][k]=1+cant[i][j][k]; calcul(j+i,0,k); } else; else
/* se toarnă din vasul 2 în vasul 1 pana acesta se umple */
    if (1+cant[i][j][k]<cant[a1][j+i-a1][k]) { cant[a1][j+i-a1][k]=1+cant[i][j][k]; calcul(a1,j+i-
a1,k); }
    if (j&&(k<a3))/* se toarnă în întregime conținutul vasului 2 în vasul 3 */
        if (k+j<=a3)
            if (1+cant[i][j][k]<cant[i][0][k+j])
                { cant[i][0][k+j]=1+cant[i][j][k]; calcul(i,0,k+j); } else; else
/* se toarnă din vasul 2 în vasul 3 pana acesta se umple */
            if (1+cant[i][j][k]<cant[i][j+k-a3][a3]) { cant[i][j+k-a3][a3]=1+cant[i][j][k]; calcul(i,j+k-a3,a3);
}
        if (k&&(i<a1))
/* se toarnă în întregime conținutul vasului 3 în vasul 1 */
        if (i+k<=a1)
            if (1+cant[i][j][k]<cant[k+i][j][0]) { cant[k+i][j][0]=1+cant[i][j][k]; calcul(k+i,j,0); }
            else; else /* se toarnă din vasul 3 în vasul 1 pana acesta se umple */
            if (1+cant[i][j][k]<cant[a1][0][i+k-a1]) { cant[a1][0][i+k-a1]=1+cant[i][j][k]; calcul(a1,0,i+k-
a1); }
        if (k&&(j<a2))/* se toarnă în întregime conținutul vasului 3 în vasul 2 */
            if (j+k<=a2)
                if (1+cant[i][j][k]<cant[i][j+k][0]) { cant[i][j+k][0]=1+cant[i][j][k]; calcul(i,j+k,0); }
                else; else /* se toarnă din vasul 3 în vasul 2 pana acesta se umple */
                if (1+cant[i][j][k]<cant[i][a2][j+k-a2])
                    { cant[i][a2][j+k-a2]=1+cant[i][j][k];
                    calcul(i,a2,j+k-a2);
                    }
            }
}
/*****/
void init(void)
{ int i,j,k;
for(i=0;i<=a1;i++) for(j=0;j<=a2;j++) for(k=0;k<=a3;k++) cant[i][j][k]=100; }
/*****/
void main(void)
{ citire(); init(); cant[a1][0][0]=0; minim=200; calcul(a1,0,0); printf("%d",minim); getch(); }

```

Probl.41. SORTAREA ARBORESCENTA. Algoritmul.

Ideea algoritmului este dispunerea elementelor șirului ca vârfuri terminale ale unui arbore binar complet. După cum se va arata în continuare, elementele șirului pot fi dispuse fie pe un nivel, fie pe doua nivele consecutive în arbore. Vom presupune ca nivelul corespunzător rădăcinii arborelui este 0.

Ne vom baza pe faptul ca, arborele binar ce-l vom construi fiind complet, pe nivelul k sunt 2^k noduri.

☐☐☐ dacă $n=2^r$, se vor dispune toate elementele șirului ca noduri terminale pe nivelul r .

☐☐☐ dacă $2^r < n < 2^{r+1}$ ($r = \lfloor \log_2 n \rfloor$) elementele șirului vor fi dispuse ca noduri terminale pe nivelele r și $r+1$, astfel:

pe nivelul $r+1$: $2n-2^{r+1}$ noduri (1)

pe nivelul r : $2^{r+1}-n$ noduri

• **Justificare**

Notam cu x numărul de noduri de pe nivelul $r+1$, respectiv cu y numărul de noduri de pe nivelul r .

Deoarece numărul total de noduri terminale este egal cu numărul elementelor șirului (n), putem scrie:

$$x+y=n \quad (2)$$

Dacă construim pentru fiecare nod de pe nivelul r cei doi fii care se găsesc pe nivelul $r+1$, obținem în total 2^{r+1} noduri (dintre care x noduri erau deja existente). Deci putem scrie:

$$x+2y=2^{r+1} \quad (3)$$

Din (2) și (3) rezulta numărul de noduri ce vor fi dispuse pe nivelele r și $r+1$, adică (1).

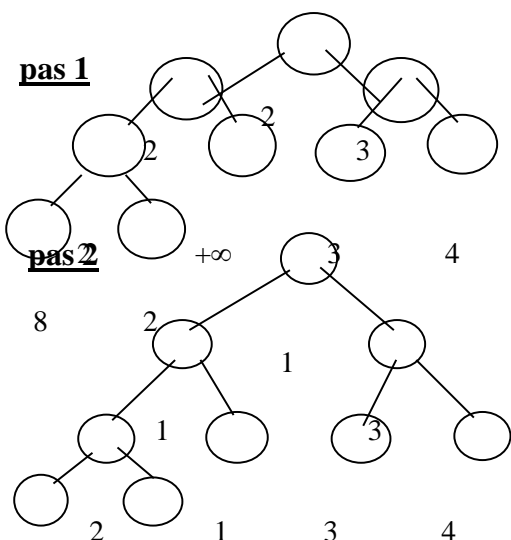
1. se vor da ca valori ale nodurilor terminale ale arborelui, elementele șirului a_1, \dots, a_n
2. până când se ajunge la rădăcina arborelui (nivelul 0), se executa
 - pentru orice doua vârfuri consecutive v_1 și v_2 de pe nivelul k ($k=r+1, 0$), se va construi un nod pe nivelul $k-1$, având ca părinți valorile minime ale valorilor nodurilor v_1 și v_2 .
3. În urma aplicării acestui procedeu, se va obține în rădăcina arborelui o valoare v , reprezentând cea mai mică valoare din arbore
4. se verifica dacă au fost obținute toate cele n elemente ale șirului
 - dacă da, STOP. S-a obținut șirul ordonat crescător
 - dacă nu, se caută nodul terminal a cărui valoare este v și se înlocuiește v cu $+\infty$ (în algoritm am folosit 32000) după care se continua cu pasul 4.

În urma aplicării acestui algoritm, se vor obține elementele șirului în ordine crescătoare.

Exemplu

pentru $n=5$ și șirul 8,2,1,3,4

- $2^2 < 5 < 2^3 \Rightarrow r=2$
- se determina $x=2 \cdot 5 - 2^3 = 2$ - noduri pe nivelul 3
 $y=5 - 2 = 3$ - noduri pe nivelul 2



$\Rightarrow 1$ este primul element în Șirul ordonat crescător

$\Rightarrow 2$ este al doilea element în Șirul ordonat crescător

După efectuarea a n pași, se va obține șirul inițial ordonat crescător: 1,2,3,4,8.

Observații

- notam șirul pe care dorim să-l sortăm x_0, x_1, \dots, x_{n-1}
- elementele șirului se citesc dintr-un fișier text

- pentru implementarea algoritmului nu am folosit o structura de arbore, ci un tablou bidimensional a, cu următoarea semnificație: $a_{k,i}$ reprezintă valoarea nodului I de pe nivelul k ($i=0, \dots, 2^k-1$) - numerotarea nodurilor pe un nivel se face de la 0.
- numărul total de comparații necesar în cazul sortării arborescente este $\leq (n-1)(\lceil \log_2 n \rceil + 1)$

```
#include<stdio.h>
#include<conio.h>
#include<iostream.h>
#include<math.h>
#include<stdlib.h>
#define maxint 32000;
int n,*x,**a,cod,nr_r1,nr_r,r;
/*****/
void citire(void)
{ FILE *pf; pf=fopen("sir.txt","r"); clrscr(); cout<<"Șirul inițial este:"<<endl; fscanf(pf,"%d",&n);
  x=new int[n];
  for(int i=0;i<n;i++) { fscanf(pf,"%d",&x[i]); cout<<x[i]<<" "; } fclose(pf); cout<<endl; }
/*****/
void dezalocare(void)
{ delete x;
  for (int i=0;i<=r;i++) delete a[i]; delete a;}
/*****/
int det(int n,int &cod) // cod=1 daca n este putere a lui 2, 0 în caz contrar
{ cod=0; double log2n=log(n)/log(2); int log2n_int=(int)log2n; cod=log2n_int==log2n?1:0; return
log2n_int; }
/*****/
void construire(void)
{ int i,j; r=det(n,cod);
  if (cod) // daca n este putere a lui 2
  { a=new int *[r+1];
    for (int i=0;i<=r;i++) a[i]=new int[(int)pow(2,i)];
    for (i=0;i<n;i++) a[r][i]=x[i]; }
  else // n nu este putere a lui 2
  { nr_r1=2*n-(int)pow(2,r+1); // numărul de noduri de pe nivelul r+1
    nr_r=(int)pow(2,r+1)-n; // numărul de noduri de pe nivelul r
    a=new int *[r+2];
    for (int i=0;i<=r;i++) a[i]=new int[(int)pow(2,i)]; a[r+1]=new int [nr_r1];
    for(i=0;i<nr_r1;i++) a[r+1][i]=x[i];
    for(j=0;j<nr_r;j++) a[r][n-(int)pow(2,r)+j]=x[nr_r1+j]; } }
/*****/
int minim(int x,int y)
{ return x<y?x:y; }
/*****/
int calcul(void)
{ int i,j;
  if (!cod) for(i=0;i<nr_r1-1;i+=2) a[r][i/2]=minim(a[r+1][i],a[r+1][i+1]);
  for(i=r;i>0;i--) for(j=0;j<(int)pow(2,i)-1;j+=2) a[i-1][j/2]=minim(a[i][j],a[i][j+1]); return a[0][0]; }
/*****/
void inlocuire(int x) // terminal - cu +∞
{ int i,j; int ct=1;
  if (cod) for (i=0;i<n;i++) if (a[r][i]==x) { a[r][i]=maxint; ct=0; }
  else; else {for(i=0;i<nr_r1;i++) if (a[r+1][i]==x) { a[r+1][i]=maxint; ct=0; }
  for(j=0;j<nr_r&&ct;j++) if (a[r][n-(int)pow(2,r)+j]==x) { a[r][n-(int)pow(2,r)+j]=maxint; ct=0;
  } } }
/*****/
void ordonare(void)
{ construire(); cout<<"Șirul ordonat este:"<<endl;
  for(int i=0;i<n;i++) { int x=calcul(); cout<<x<<" "; inlocuire(x); } }
```

/*****/

void main(void)

{ citire(); ordonare(); dezalocare(); getch(); }

PROBLEME PROPUSE SUPLIMENTAR

1. Se cere sa se genereze toate descompunerile numărului natural n ca suma de numere naturale (doua partiții diferă fie prin valorile elementelor din partiție, fie prin ordinea acestora)
2. Sa se genereze toate matricele $n \times n$ ce conțin elemente distincte din mulțimea $\{1, \dots, n^2\}$, astfel încât nici un element din matrice sa nu aibă aceeași paritate cu vecinii săi (vecinii unui element se considera în direcțiile N,S,E,V).
3. Pe suprafața unei tari se găsesc n râuri. Se citesc de la intrare perechi de forma : i, j , cu următoarea semnificație: râul i este afluent al râului j . Se numește *Debitul* izvorului i , cantitatea de apa care trece prin secțiunea izvorului râului i în unitatea de timp. Debitul râului i la vărsare va fi egal cu *Debitul* izvorului i plus suma debitelor afluenților la vărsare în râul i . Dându-se pentru fiecare râu debitul izvorului sau, sa se calculeze debitul la vărsare al fiecărui râu.
4. Un țaran primește o bucata dreptunghiulara de pământ pe care dorește sa planteze o livada. Pentru aceasta el va împărți bucata de pământ în $n \times m$ pătrate, având dimensiunile egale, iar în fiecare pătrat va planta un singur pom din următoarele soiuri pe care le are la dispoziție: cais, nuc, mar și prun. Sa se afișeze toate variantele de a alcătui livada respectând următoarele condiții:
Nu trebuie sa existe doi pomi de același soi în doua căsuțe învecinate pe orizontala, verticala sau diagonală.
Fiecare pom va fi înconjurat de cel puțin un pom din toate celelalte trei soiuri.
Observație : taranul are la dispoziție suficienți pomi de fiecare soi.
5. Se citește dintr-un fișier un text ce se termina cu punct și conține litere iar ca și separator blankul. Sa se construiască șirul l_1, \dots, l_k cu literele distincte din text și șirul f_1, \dots, f_k al frecvențelor de apariție al acestor litere în text. Sa se introducă un număr maxim de elemente din șirul f într-o matrice pătratică, astfel încât: elementele din matrice sa fie distincte nici un element din matrice sa nu aibă aceeași paritate cu vecinii săi (N, S, E, V)
6. Un cal și un rege se afla pe o tabla pătratică de dimensiune $n \times n$. Unele poziții sunt “arse”, pozițiile lor fiind cunoscute. Calul nu poate calca pe câmpuri “arse”, iar orice mișcare a calului face ca respectivul câmp și toți vecinii acestuia (N, S, E, V) sa devina “arse”. Sa se afle daca exista o succesiune de mutări permise (cu restricțiile de mai sus) prin care calul sa poată ajunge la rege și sa revină în poziția inițială. Poziția inițială a regelui, precum și poziția calului sunt “nearse”.
7. Se da un lac înghețat sub forma unui tablou bidimensional $m \times n$. O broscuță se găsește în poziția (1,1), iar în poziția (m,n) se găsește o insecta. În gheata se găsesc găuri ale căror coordonate se cunosc. Găsiți drumul cel mai scurt al broscuței până la insecta și înapoi, știind ca:
deoarece broscuță visa sa devina cal, ea va sari ca un cal (de șah) în punctele în care pășește broscuță, gheata cedează broscuță nu poate pași decât pe gheata
8. Se considera o tabla de șah de dimensiune $n \times n$ pe care sunt dispuse obstacole. Se cere sa se tipărească numărul minim de mutări necesare unui nebun pentru a se deplasa, respectând regulile jocului de șah și ocolind obstacolele, dintr-o poziție inițială data într-o poziție finala data. Se considera ca în poziția inițială și cea finala a nebunului nu exista obstacole.
9. Se da un labirint sub forma unei matrice $m \times n$. Anumite poziții ale labirintului sunt ocupate. În doua poziții precizate se găsesc doi îndrăgostiți, Romeo și Julieta. Se cere sa se găsească drumul de lungime minima pe care trebuie sa-l străbată cei doi îndrăgostiți pentru a se întâlni, știind ca nu poate trece decât prin poziții libere. Se va presupune ca pozițiile inițiale ale celor doi îndrăgostiți sunt libere.
10. Pe o matrice pătratică de dimensiune $n \times n$ sunt dispuse m pietre. Pietrele se pot mișca pe matrice astfel: o piatra de pe poziția (i,j) se poate deplasa în poziția (i-2,j) (respectiv (i,j-2), (i+2,j), (i,j+2)) doar daca în poziția (i-1,j) (respectiv (i,j-1), (i+1,j), (i,j+1)) exista o piatra, care în urma deplasării dispăre. Știind ca la un moment dat se poate deplasa orice piatra din matrice, se cere sa se precizeze numărul minim de mutări astfel încât în final pe matrice sa rămână o singura piatra.
11. Sa se înmulțească 2 numere mari folosind tehnica DIVIDE ET IMPERA.
12. Sa se implementeze algoritmul de căutare binară într-un sir x_1, \dots, x_n ordonat crescător. Dându-se o valoare v sa se precizeze daca aceasta valoare se afla sau nu printre componentele șirului, în caz afirmativ sa se precizeze poziția pe care apare în sir.
13. Sa se simuleze următorul joc între doi parteneri (de exemplu între o persoana și calculator). Jucătorul 2 trebuie sa ghicească un număr natural (cuprins între 1 și 32700) ascuns de jucătorul 1. Atunci când jucătorul 2 propune un număr I se răspunde prin:
1 daca numărul este prea mare
-1 daca numărul este prea mic
0 daca numărul a fost găsit

14. Se dau 2 șiruri de numere a_1, a_2, \dots, a_n și b_1, b_2, \dots, b_m . Se cere să se găsească cel mai lung subșir comun al celor două șiruri.

Se dau două șiruri de caractere. Se cere să se transforme al doilea șir în primul folosind cât mai puține operații, o operație putând fi una din următoarele:

se înlocuiește o literă din al doilea șir cu o literă din primul șir

se inserează o literă în al doilea șir

se șterge o literă din al doilea șir

15. O galerie de pictură are $n \times m$ camere egale. Fiecare din ele are uși de comunicare cu camerele vecine. La intrarea în fiecare camera un vizitator platește o anumită sumă de bani, în funcție de valoarea picturilor din camera. Intrarea în galerie se face numai pe ușa $(1,1)$ iar ieșirea este unica: prin camera (n,m) . Se cere să se determine cel mai scump traseu.

Fie funcțiile f, g definite pentru numere întregi astfel: $f(a) = \lfloor a/2 \rfloor$ și $g(a) = 3a$. Să se determine din câte aplicări ale funcțiilor f și g se poate ajunge de la numărul X la numărul Y , ambele date inițial.

16. Se considera o tablă de $n \times n$ pe care sunt dispuse obstacole. Se cere să se tipărească numărul minim de mutări necesare unui nebun pentru a se deplasa, respectând regulile jocului de $n \times n$ și ocolind obstacolele, dintr-o poziție inițială dată într-o poziție finală dată.

17. Un om dorește să urce o scară cu N trepte. El poate urca la un moment dat una sau două trepte. Precizați în câte moduri poate urca omul scara.

18. Problema Se dă un vector cu n componente la început nule. O secțiune pe poziția k va incrementa toate elementele aflate în zona de secționare anterioară situate între poziția 1 și k . **Exemplu.**

0 0 0 0 0 0 se secționează pe poziția 4;

1 1 1 1 0 0 se secționează pe poziția 1;

2 1 1 1 0 0 se secționează pe poziția 3;

3 2 2 1 0 0 etc.

Să se determine o ordine de secționare a unui vector cu n elemente astfel încât suma elementelor sale să fie s . Valorile n și s se citesc.

19. Problema Presupunând ca 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.

iii) Presupunând ca există monezi de: k^0, k^1, \dots, k^{n-1} unități, pentru $k \in \mathbb{N}, k > 1$ oarecare, arătați că metoda greedy da mereu soluția optimă. Considerați că n este un număr finit și că din fiecare tip de monedă există o cantitate nelimitată.

20. Problema Pe o bandă magnetică sunt n programe, un program i de lungime l_i fiind apelat cu probabilitatea p_i , $1 \leq i \leq n$, $p_1 + p_2 + \dots + p_n = 1$. 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 .

21. Problema 30. 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.

22. Problema 31. Scrieți algoritmul greedy pentru colorarea unui graf și analizați eficiența lui.

23. Problema 32. Ce se întâmplă cu algoritmul greedy din problema comis-voiajorului dacă admitem că pot exista două orașe fără legătura directă între ele?

24. Problema 33. 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.

25 Problema 34. Codul Fibonacci de ordinul n , $n \geq 2$, este secvență 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: $C_2 = (\lambda)$, $C_3 = (0, \lambda)$, $C_4 = (00, 01, \lambda)$, $C_5 = (000, 001, 010, 100, \lambda)$ 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 ca putem construi codul Fibonacci de ordinul n , $n \geq 4$, prin concatenarea a doua subsecvente. Prima subsecventa se obține prefixând cu 0 fiecare codificare din C_{n-1} . A doua subsecventa se obține prefixând cu 10 fiecare codificare din C_{n-2} .

BIBLIOGRAFIE

1. Doina Logofătu: *Algoritmi fundamentali în C++*. Aplicații, Ed. 1, Editura Polirom, Iași, 2007, [ISBN 9734600939](#).
2. ILIE GARBACEA, RAZVAN ANDONE, *Algoritmi fundamentali, o perspectiva C++*. Editura Polirom, Iași, 2006
3. Хусаинов Б.С. Структуры и алгоритмы обработки данных. Примеры на языке Си: Учеб. пособие. – Финансы и статистика, 2004.
4. Х.М. Дейтел, П. Дж. Дейтел. *Как программировать на C++: Четвёртое издание. Пер. с англ.*— М.: ООО «Бином—Пресс», 2005г.
5. Дж. Макконнелл. Основы современных алгоритмов. 2-е дополненное издание. Москва: Техносфера, 2004.
6. Морозов М. Большая книга загадок и головоломок № 5. Эгмонт Россия Лтд, 2004.
http://www.emis.de/journals/short_index.html
<http://www.ai.mit.edu>
<http://www.cs.berkeley.edu/~russell/aima.html>
<http://www.cs.umbc.edu/471/lectures/>
<http://ai.about.com/compute/software/ai/gi/dynamic/offsite.htm?site=http://www.cs.vu.nl/~victor/thesis.html>