PROF.UNIV.DR. HORIA IOAN GEORGESCU

TEHNICI AVANSATE DE PROGRAMARE

1 DESPRE ALGORITMI

1.1. Diferențe între informatică şi matematică

Este de multă vreme acceptat, dar de puțină vreme recunoscut şi statuat în România, faptul că informatica şi matematica sunt două ştiințe de-sine-stătătoare.

Aceasta nu înseamnă însă că informatica nu foloseşte instrumentul matematic! Din contră, instrumentul matematic este esențial în informatică. Dincolo de nevoia de a demonstra diferite aserțiuni (în care intervine matematica, dar numai prin anumite capitole!), este vorba de a avea o privire de asamblu ordonată şi coerentă asupra informaticii. Fără această viziune care să cuprindă elementele esențiale, vom avea mereu impresia că degeaba învățăm astăzi unele aspecte, pentru că peste doi ani ele vor fi demodate şi nefolositoare. În fapt, anumite aspecte sunt de durată şi însuşirea lor ne va ajuta să facem față fluxului necontenit de informații şi produse ce apar pe piață.

Vom prezenta numai două aspecte ce deosebesc cele două ştiințe:

1) *În lucrul pe calculator, majoritatea proprietăților algebrice nu sunt satisfăcute*.

- *elementul neutru*: egalitatea `a+b=a` poate fi satisfăcută fără ca `b=0`: este situația în care `b≠0`, dar ordinul său de mărime este mult mai mic decât al lui `a`, astfel încât memorarea unui număr limitat de cifre şi aducerea la acelaşi exponent fac ca, la efectuarea adunării, `b` să fie considerat egal cu 0.
- *comutativitate*: să considerăm următoarea funcție scrisă în Pascal ce foloseşte apelul prin referință:

```
function f(var a:integer):integer;
begin a:=a+1; f:=a end;
```

Secvența de instrucțiuni:

```
a:=1; write (a+f(a))
```

produce la ieşire valoarea 1+2=3, pe când secvența de instrucțiuni:

```
a:=1; write (f(a)+a))
```

produce la ieşire valoarea 2+2=4.

- *asociativitate*: pentru simplificare (dar fără a reduce din generalitate) vom presupune că numerele sunt memorate cu o singură cifră semnificativă şi că la înmulţire se face trunchiere. Atunci rezultatul înmulţirilor $(0.5\times0.7)\times0.9$ este $0.3\times0.9=0.2$, pe când rezultatul înmulţirilor $0.5\times(0.7\times0.9)$ este $0.5\times0.6=0.3$.

2) *Nu interesează în general demonstrarea teoretică a existenţei unei soluţii, ci accentul este pus pe elaborarea algoritmilor.*

Vom pune în vedere acest aspect prezentând o elegantă demonstraţie a următoarei propoziţii:

Propoziţie. Există $\alpha,\beta\in\mathbf{R}\setminus\mathbf{Q}$ cu $\alpha^{\beta}\in\mathbf{Q}$.

Pentru demonstraţie să considerăm numărul real $x=a^a$, unde $a=\sqrt{2}$.

Dacă $x\in\mathbf{Q}$, propoziţia este demonstrată.

Dacă $x\notin\mathbf{Q}$, atunci $x^a=a^2=2\in\mathbf{Q}$ şi din nou propoziţia este demonstrată.

Frumuseţea acestei scurte demonstraţii nu poate satisface pe un informatician, deoarece lui i se cere în general să furnizeze un rezultat şi nu să arate existenţa acestuia.

1.2. Aspecte generale care apar la rezolvarea unei probleme

Aşa cum am spus, informaticianului i se cere *să elaboreze un algoritm* pentru o problemă dată, care să furnizeze o soluţie (fie şi aproximativă, cu condiţia menţionării acestui lucru).

Teoretic, paşii sunt următorii:

1) demonstrarea faptului că este posibilă elaborarea unui algoritm pentru determinarea unei soluţii;
2) elaborarea unui algoritm (caz în care pasul anterior devine inutil);
3) demonstrarea corectitudinii algoritmului;
4) determinarea timpului de executare a algoritmului;
5) demonstrarea optimalităţii algoritmului (a faptului că timpul de executare este mai mic decât timpul de executarea al oricărui alt algoritm pentru problema studiată).

Evident, acest scenariu este idealist, dar el trebuie să stea în permanenţă în atenţia informaticianului.

În cele ce urmează, vom schiţa câteva lucruri legate de aspectele de mai sus, nu neapărat în ordinea menţionată.

1.3. Timpul de executare a algoritmilor

Un algoritm este elaborat nu numai pentru un set de date de intrare, ci pentru o mulțime de astfel de seturi. De aceea trebuie bine precizată mulțimea (seturilor de date) de intrare. Timpul de executare se măsoară în funcție de lungimea `n` a datelor de intrare.

Ideal este să determinăm o formulă matematică pentru `T(n)`=timpul de executare pentru orice set de date de intrare de lungime `n`. Din păcate, acest lucru nu este în general posibil. De aceea, în majoritatea cazurilor ne mărginim la a evalua *ordinul de mărime* al timpului de executare.

Mai precis, spunem că timpul de executare este de ordinul `f(n)` şi exprimăm acest lucru prin `T(n)=O(f(n))`, dacă raportul între `T(n)` şi `f(n)` tinde la un număr real atunci când `n` tinde la ∞.

De exemplu, dacă $f(n)=n^k$ pentru un anumit număr `k`, spunem că algoritmul este *polinomial*.

Specificăm, fără a avea pretenția că dăm o definiție, că un algoritm se numeşte "acceptabil" dacă este este polinomial. În capitolul referitor la metoda backtracking este prezentat un mic tabel care scoate în evidență faptul că algoritmii exponențiali necesită un timp de calcul mult prea mare chiar pentru un `n` mic şi indiferent de performanțele calculatorului pe care lucrăm.

De aceea, în continuare accentul este pus pe prezentarea unor algoritmi polinomiali.

1.4. Corectitudinea algoritmilor

În demonstrarea corectitudinii algoritmilor, există două aspecte importante:

- *Corectitudinea parțială*: presupunând că algoritmul se termină (într-un număr finit de paşi), trebuie demonstrat că rezultatul este corect;
- *Terminarea programului*: trebuie demonstrat că algoritmul se încheie în timp finit.

Evident, condițiile de mai sus trebuie îndeplinite pentru *orice* set de date de intrare admis.

Modul tipic de lucru constă în introducerea în anumite locuri din program a unor *invarianți*, adică relații ce sunt îndeplinite la orice trecere a programului prin acele locuri.

Ne mărginim la a prezenta două exemple simple.

Exemplul 1. Determinarea concomitentă a celui mai mare divizor comun şi a celui mai mic multiplu comun a două numere naturale.

Fie $a,b\in\mathbf{N}^*$. Se caută:

- (a,b)=cel mai mare divizor comun al lui a şi b;
- $[a,b]$=cel mai mic multiplu comun al lui a şi b.

Algoritmul este următorul:

```
x ← a; y ← b; u ← a; v ← b;
while x≠y
   { xv+yu = 2ab; (x,y)=(a,b) }                    (*)
   if x>y then x ← x-y; u ← u+v
          else y ← y-x; v ← u+v
write(x,(u+v)/2)
```

Demonstrarea corectitudinii se face în trei paşi:

1) $(*)$ *este invariant*:

La prima intrare în ciclul `while`, condiţia este evident îndeplinită.

Mai trebuie demonstrat că dacă relaţiile (*) sunt îndeplinite şi ciclul se reia, ele vor fi îndeplinite şi după reluare.

Fie (x,y,u,v) valorile curente la o intrare în ciclu, iar (x',y',u',v') valorile curente la următoarea intrare în ciclul `while`. Deci: $xv+yu=2ab$ şi $(x,y)=(a,b)$.

Presupunem că $x>y$. Atunci $x'=x-y$, $y'=y$, $u'=u+v$, $v'=v$.

$x'v'+y'u'=(x-y)v+y(u+v)=xv+yu=2ab$ şi

$(x',y')=(x-y,y)=(x,y)=(a,b)$.

Cazul $x<y$ se studiază similar.

2) *Corectitudinea parţială*:

La ieşirea din ciclul `while`, $x=(x,x)=(x,y)=(a,b)$. Notăm $d=(a,b)=x$. Conform relaţiilor $(*)$, avem $d(u+v)=2\alpha\beta d^2$, unde $a=\alpha d$ şi $b=\beta d$. Atunci $(u+v)/2=\alpha\beta d=ab/d=[a,b]$.

3) *Terminarea programului*:

Fie $\{x_n\}$, $\{y_n\}$, $\{u_n\}$, $\{v_n\}$ şirul de valori succesive ale variabilelor. Toate aceste valori sunt numere naturale pozitive. Se observă că şirurile $\{x_n\}$ şi $\{y_n\}$ sunt descrescătoare, iar şirul $\{x_n+y_n\}$ este *strict* descrescător. Aceasta ne asigură că după un număr finit de paşi vom obţine $x=y$.

Exemplul 2. Metoda de înmulţire a ţăranului rus.

Fie $a,b\in\mathbf{N}$. Se cere să se calculeze produsul ab.

Ţăranul rus ştie doar:

- să verifice dacă un număr este par sau impar;
- să adune două numere;
- să afle câtul împărţirii unui număr la 2;
- să compare un număr cu 0.

Cu aceste cunoştinţe, ţăranul rus procedează astfel:

```
x ← a; y ← b; p ← 0
while x>0
   { xy+p=ab }                                    (*)
   if x impar then p ← p+y
   x ← x div 2; y ← y+y
write(p)
```

Să urmărim cum decurg calculele pentru $x=54$, $y=12$:

x	y	p
54	12	0
27	24	
13	48	24
6	96	72
3	192	
1	384	264
0	?	648

Ca şi pentru exemplul precedent, demonstrarea corectitudinii se face în trei paşi:

1) $(*)$ *este invariant*:

La prima intrare în ciclul `while`, relaţia este evident îndeplinită.

Mai trebuie demonstrat că dacă relaţia (*) este îndeplinită şi ciclul se reia, ea va fi îndeplinită şi la reluare.

Fie (x,y,p) valorile curente la o intrare în ciclu, iar (x',y',p') valorile curente la următoarea intrare în ciclul `while`. Deci: $xy+p=ab$.

Presupunem că x este impar. Atunci $(x',y',p')=((x-1)/2,2y,p+y)$. Rezultă $x'y'+p'=(x-1)/2\cdot 2y+p+y=xy+p=ab$.

Presupunem că x este par. Atunci $(x',y',p')=(x/2,2y,p)$. Rezultă $x'y'+p'=xy+p=ab$.

2) *Corectitudinea parţială*:

Dacă programul se termină, atunci $x=0$, deci $p=ab$.

3) *Terminarea programului*:

Fie $\{x_n\}$, $\{y_n\}$ şirul de valori succesive ale variabilelor corespunzătoare.

Se observă că şirul $\{x_n\}$ este *strict* descrescător. Aceasta ne asigură că după un număr finit de paşi vom obţine $x=0$.

1.5. Optimalitatea algoritmilor

Să presupunem că pentru o anumită problemă am elaborat un algoritm şi am putut calcula şi timpul său de executare $T(n)$. Este natural să ne întrebăm dacă algoritmul nostru este "cel mai bun" sau există un alt algoritm cu timp de executare mai mic.

Problema demonstrării optimalității unui algoritm este foarte dificilă, în mod deosebit datorită faptului că trebuie să considerăm *toți* algoritmii posibili şi să arătăm că ei au un timp de executare superior.

Ne mărginim la a enunța două probleme şi a demonstra optimalitatea algoritmilor propuşi, pentru a pune în evidență dificultățile care apar.

Exemplul 3. Determinarea celui mai mic element al unui vector.

Se cere să determinăm $m=\min(a_1,a_2,\ldots,a_n)$.

Algoritmul binecunoscut este următorul:

$m \leftarrow a_1$
for i=2,n
   if $a_i<m$ then $m \leftarrow a_i$

care necesită $n-1$ comparări între elementele vectorului $a=(a_1,a_2,\ldots,a_n)$.

Propoziția 1. Algoritmul de mai sus este optimal.

Trebuie demonstrat că orice algoritm bazat pe comparări necesită cel puțin $n-1$ comparări.

Demonstrarea optimalității acestui algoritm se face uşor prin inducție.

Pentru $n=1$ este evident că nu trebuie efectuată nici o comparare.

Presupunem că orice algoritm care rezolvă problema pentru n numere efectuează cel puțin $n-1$ comparări şi să considerăm un algoritm oarecare care determină cel mai mic dintre $n+1$ numere. Considerăm prima comparare efectuată de acest algoritm; fără reducerea generalității, putem presupune că s-au comparat a_1 cu a_2 şi că $a_1<a_2$. Atunci $m=\min(a_1,a_3,\ldots,a_{n+1})$. Dar pentru determinarea acestui minim sunt necesare cel puțin $n-1$ comparări, deci numărul total de comparări efectuat de algoritmul considerat este cel puțin egal cu n.

Exemplul 4. Determinarea minimului şi maximului elementelor unui vector.

Se cere determinarea valorilor $m=\min(a_1,a_2,\ldots,a_n)$ şi $M=\max(a_1,a_2,\ldots,a_n)$.

Determinarea succesivă a valorilor m şi M necesită timpul $T(n)=2(n-1)$.

O soluție mai bună constă în a considera câte două elemente ale vectorului, a determina pe cel mai mic şi pe cel mai mare dintre ele, iar apoi în a compara pe cel mai mic cu minimul curent şi pe cel mai mare cu maximul curent:

```
if n impar then m ← a1; M ← a1; k ← 1
          else if a1<a2 then m ← a1; M ← a2
                        else m ← a2; M ← a1
               k ← 2
{ k = numărul de elemente analizate }
while k≤n-2
   if ak+1<ak+2 then if ak+1<m then m ← ak+1
                     if ak+2>M then M ← ak+2
                else if ak+2<m then m ← ak+2
                     if ak+1>M then M ← ak+1
   k ← k+2
```

Să calculăm numărul de comparări efectuate:

- pentru $n=2k$, în faza de inițializare se face o comparare, iar în continuare se fac $3(k-1)$ comparări; obținem $T(n)=1+3(k-1)=3k-3=3n/2-2=\lceil 3n/2\rceil-2$.
- pentru $n=2k+1$, la inițializare nu se face nici o comparare, iar în continuare se fac $3k$ comparări; obținem $T(n)=(3n-3)/2=(3n+1)/2-2=\lceil 3n/2\rceil-2$.

În concluzie, timpul de calcul este $T(n)=\lceil 3n/2\rceil-2$.

Propoziția 2. Algoritmul de mai sus este optimal.

Considerăm următoarele mulțimi şi cardinalul lor:

- A= mulțimea elementelor care nu au participat încă la comparări; $a=|A|$;
- B= mulțimea elementelor care au participat la comparări şi au fost totdeauna mai mari decât elementele cu care au fost comparate; $b=|B|$;
- C= mulțimea elementelor care au participat la comparări şi au fost totdeauna mai mici decât elementele cu care au fost comparate; $c=|C|$;
- D= mulțimea elementelor care au participat la comparări şi au fost cel puțin o dată mai mari şi cel puțin o dată mai mici decât elementele cu care au fost comparate; $d=|D|$.

Numim *configurație* un quadruplu (a,b,c,d). Problema constă în determinarea numărului de comparări necesare pentru a trece de la quadruplul $(n,0,0,0)$ la quadruplul $(0,1,1,n-2)$.

Considerăm un algoritm arbitrar care rezolvă problema şi arătăm că el efectuează cel puțin $\lceil 3n/2\rceil-2$ comparări.

Să analizăm trecerea de la o configurație oarecare (a,b,c,d) la următoarea. Este evident că nu are sens să efectuăm comparări în care intervine vreun element din D. Apar următoarele situații posibile:

1) Compar două elemente din A: se va trece în configurația (a-2,b+1,c+1,d) .
2) Compar două elemente din B: se va trece în configurația (a,b-1,c,d+1) .
3) Compar două elemente din C: se va trece în configurația (a,b,c-1,d+1) .
4) Se compară un element din A cu unul din B. Sunt posibile două situații:
   - elementul din A este mai mic: se trece în configurația (a-1,b,c+1,d);
   - elementul din A este mai mare: se trece în configurația (a-1,b,c,d+1).

   Cazul cel mai defavorabil este primul, deoarece implică o deplasare "mai lentă" spre dreapta a componentelor quadruplului. De aceea vom lua în considerare acest caz.
5) Se compară un element din A cu unul din C. Sunt posibile două situații:
   - elementul din A este mai mic: se trece în configurația (a-1,b,c,d+1);
   - elementul din A este mai mare: se trece în configurația (a-1,b+1,c,d).

   Cazul cel mai defavorabil este al doilea, deoarece implică o deplasare "mai lentă" spre dreapta a componentelor quadruplului. De aceea vom lua în considerare acest caz.
6) Se compară un element din B cu unul din C. Sunt posibile două situații:
   - elementul din B este mai mic: se trece în configurația (a,b-1,c-1,d+2);
   - elementul din B este mai mare: se rămâne în configurația (a,b,c,d).

   Cazul cel mai defavorabil este al doilea, deoarece implică o deplasare "mai lentă" spre dreapta a componentelor quadruplului. De aceea vom lua în considerare acest caz.

Observație. Cazurile cele mai favorabile sunt cele în care d creşte, deci ies din calcul elemente candidate la a fi cel mai mic sau cel mai mare.

Odată stabilită trecerea de la o configurație la următoarea, ne punem problema cum putem trece mai rapid de la configurația inițială la cea finală.

Analizăm cazul în care n=2k (cazul în care n este impar este propus ca exercițiu). Trecerea cea mai rapidă la configurația finală se face astfel:

- plecăm de la (n,0,0,0)=(2k,0,0,0);
- prin k comparări între perechi de elemente din A ajungem la (0,k,k,0);
- prin k-1 comparări între perechi de elemente din B ajungem la (0,1,k,k-1);
- prin k-1 comparări între perechi de elemente din C ajungem la (0,1,1,n-2).

În total au fost necesare k+(k-1)+(k-1)=3k-2=$\lceil 3n/2 \rceil$-2 comparări.

1.6. Existența algoritmilor

Acest aspect este şi mai delicat decât precedentele, pentru că necesită o definiție matematică riguroasă a noțiunii de algoritm. Nu vom face decât să prezentăm (fără vreo demonstrație) câteva definiții şi rezultate. Un studiu mai amănunțit necesită un curs aparte!

Începem prin a preciza că problema existenței algoritmilor a stat în atenția matematicienilor încă înainte de apariția calculatoarelor. În sprijinul acestei afirmații ne rezumăm la a spune că un rol deosebit în această teorie l-a jucat matematicianul englez Alan Turing (1912-1954), considerat părintele inteligenței artificiale.

Deci *ce este un algoritm*?

Noțiunea de algoritm nu poate fi definită decât pe baza unui limbaj sau a unei maşini matematice abstracte.

Prezentăm în continuare o singură definiție, care are la bază *limbajul S* care operează asupra numerelor naturale.

Un program în limbajul *S* foloseşte variabilele:

- $x_1, x_2, \ldots$ care constituie datele de intrare (nu există instrucțiuni de citire);
- y (în care va apărea rezultatul prelucrărilor);
- $z_1, z_2, \ldots$ care constituie variabile de lucru.

Variabilele $y, z_1, z_2, \ldots$ au inițial valoarea 0.

Instrucțiunile pot fi etichetate (nu neapărat distinct) şi au numai următoarele forme, în care v este o variabilă, iar L este o etichetă:

- $v \leftarrow v+1$ { valoarea variabilei v creşte cu o unitate }
- $v \leftarrow v-1$ { valoarea variabilei v scade cu o unitate dacă era strict pozitivă }
- `if v>0 goto L` { se face transfer condiționat la prima instrucțiune cu eticheta L, dacă o astfel de instrucțiune există; în caz contrar programul se termină }.

Programul se termină fie dacă s-a executat ultima instrucțiune din program, fie dacă se face transfer la o instrucțiune cu o etichetă inexistentă.

Observații:

- faptul că se lucrează numai cu numere naturale nu este o restricție, deoarece în memorie există doar secvențe de biți (interpretate în diferite moduri);
- nu interesează timpul de executare a programului, ci numai existența sa;

- dacă rezultatul dorit constă din mai multe valori, vom scrie câte un program pentru calculul fiecăreia dintre aceste valori;
- programul vid corespunde calculului funcției identic egală cu 0: pentru orice $x_1, x_2, \ldots$ valoarea de ieşire a lui y este 0 (cea inițială).

Este naturală o neîncredere inițială în acest limbaj, dacă îl comparăm cu limbajele evoluate din ziua de azi. Se poate însă demonstra că în limbajul *S* se pot efectua calcule "oricât" de complexe asupra numerelor naturale.

***Teza lui Church* (1936).** *Date fiind numerele naturale* $x_1, x_2, \ldots, x_n$, *numărul* y *poate fi "calculat" pe baza lor dacă şi numai dacă există un program în limbajul S care pentru valorile de intrare* $x_1, x_2, \ldots, x_n$ *produce la terminarea sa valoarea* y.

Cu alte cuvinte, înțelegem prin ***algoritm*** ce calculează valoarea y plecând de la valorile $x_1, x_2, \ldots, x_n$ un program în limbajul *S* care realizează acest lucru.

Există mai multe definiții ale noțiunii de algoritm, bazate fie pe calculul cu numere naturale fie pe calcul simbolic, folosind fie limbaje de programare fie maşini matematice, dar toate s-au dovedit a fi echivalente (cu cea de mai sus)!

Mai precizăm că orice program în limbajul *S* poate fi codificat ca un număr natural (în particular mulțimea acestor programe este numărabilă).

Numim *problemă nedecidabilă* o problemă pentru care nu poate fi elaborat un algoritm. Definirea matematică a noțiunii de algoritm a permis detectarea de probleme nedecidabile. Câteva dintre ele sunt următoarele:

1) *Problema opririi programelor*: pentru orice program şi orice valori de intrare să se decidă dacă programul se termină.
2) *Problema opririi programelor* (*variantă*): pentru un program dat să se decidă dacă el se termină pentru orice valori de intrare.
3) *Problema echivalenței programelor*: să se decidă pentru orice două programe dacă sunt echivalente (produc aceeaşi ieşire pentru aceleaşi date de intrare).

În continuare vom lucra cu noțiunea de algoritm în accepțiunea sa uzuală, aşa cum în liceu (şi la unele facultăți) se lucrează cu numerele reale, fără a se prezenta o definiție riguroasă a lor.

Am dorit însă să evidențiem că există probleme nedecidabile şi că (uimitor pentru unii) studiul existenței algoritmilor a început înainte de apariția calculatoarelor.

2 ARBORI

2.1. Grafuri

Numim *graf neorientat* o pereche $G=(V,M)$, unde:
- V este o mulțime finită şi nevidă de elemente numite *vârfuri* (*noduri*);
- M este o mulțime de perechi de elemente distincte din V, numite muchii. O muchie având vârfurile i şi j (numite *extremităţile* sale) este notată prin (i,j) sau (j,i).

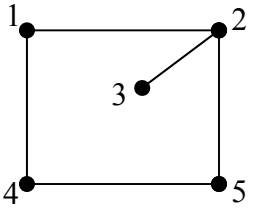
Un *subgraf* al lui G este un graf $G'=(V',M')$ unde $V'\subset V$, iar M' este formată din toate muchiile lui G care unesc vârfuri din V'.

Un g*raf parţial* al lui G este un graf $G'=(V,M')$ cu $M'\subset M$.

Numim *drum* o succesiune de muchii $(i_1,i_2),(i_2,i_3),\ldots,(i_{k-1},i_k)$, notată şi prin $(i_1,i_2,\ldots,i_k)$. Dacă toate vârfurile drumului sunt distincte, atunci el se numeşte *drum elementar*.

Un *ciclu elementar* este un drum $(i_1,i_2,\ldots,i_k,i_1)$, cu $(i_1,i_2,\ldots,i_k)$ drum elementar şi $k\geq 3$. Un *ciclu* este un drum $(i_1,i_2,\ldots,i_k,i_1)$ care conţine cel puţin un ciclu elementar.

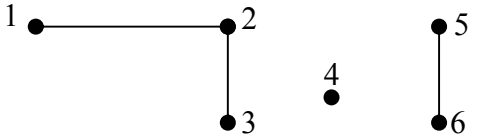
Exemplul 1. În graful alăturat:
- (1,2,5,4) este un drum de la 1 la 4;
- (5,2,1,4,5) este un ciclu elementar;
- (1,2,1) nu este un ciclu.



Un graf neorientat se numeşte *conex* dacă oricare două vârfuri ale sale sunt unite printr-un drum.

În cazul unui graf neconex, se pune problema determinării componentelor sale conexe; o *componentă conexă* este un subgraf conex maximal. Descompunerea în componente conexe determină atât o partiţie a vârfurilor, cât şi a muchiilor.

Exemplul 2. Graful următor are două componente conexe şi anume subgrafurile determinate de submulţimile de vârfuri {1,2,3}, {4}, {5,6}.



În cele ce urmează vom nota prin `n` numărul vârfurilor, iar prin `m` numărul muchiilor unui graf: `n=|V|`, `m=|M|`. În analiza complexității în timp a algoritmilor pe grafuri, aceasta va fi măsurată în funcție de `m+n`, adică se va ține cont atât de numărul vârfurilor cât şi de cel al muchiilor.

Prin gradul `grad(i)` al unui vârf `i` înțelegem numărul muchiilor care îl au ca extremitate.

Un *graf orientat* este tot o pereche `G=(V,M)`, deosebirea față de grafurile neorientate constând în faptul că elementele lui `M` sunt perechi *ordonate* de vârfuri numite *arce*; altfel spus, orice arc `(i,j)` are stabilit un sens de parcurgere şi anume de la extremitatea sa inițială `i` la extremitatea sa finală `j`.

Noțiunile de drum, drum elementar, ciclu şi ciclu elementar de la grafurile neorientate se transpun în mod evident la grafurile orientate, cu singura observație că în loc de ciclu vom spune *circuit*.

Pentru un vârf `i`:

- gradul interior $grad^-(i)$ este numărul arcelor ce sosesc în `i`;
- gradul exterior $grad^+(i)$ este numărul arcelor ce pleacă din `i`.

2.2. Arbori

Numim *arbore* un graf neorientat conex şi fără cicluri.

Aceasta nu este singurul mod în care putem defini arborii. Câteva definiții echivalente apar în următoarea teoremă, expusă fără demonstrație.

Teoremă. Fie `G` un graf cu $n \geq 1$ vârfuri. Următoarele afirmații sunt echivalente:

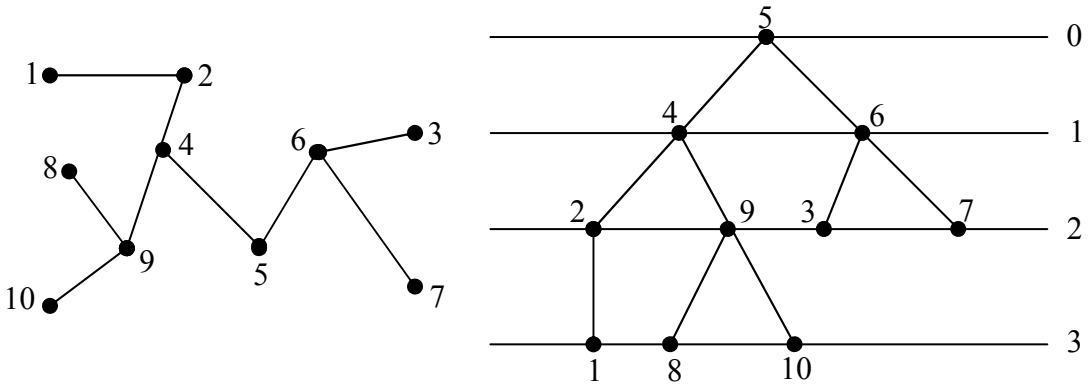
1) `G` este un arbore;
2) `G` are `n-1` muchii şi nu conține cicluri;
3) `G` are `n-1` muchii şi este conex;
4) oricare două vârfuri din `G` sunt unite printr-un unic drum;
5) `G` nu conține cicluri şi adăugarea unei noi muchii produce un unic ciclu elementar;
6) `G` este conex, dar devine neconex prin ştergerea oricărei muchii.

În foarte multe probleme referitoare la arbori este pus în evidență un vârf al său, numit *rădăcină*. Alegerea unui vârf drept rădăcină are două consecințe:

- *Arborele poate fi aşezat pe niveluri* astfel:
- rădăcina este aşezată pe nivelul 0;
- pe fiecare nivel `i` sunt plasate vârfurile pentru care lungimea drumurilor care le leagă de rădăcină este `i`;
- se trasează muchiile arborelui.

Această aşezare pe niveluri face mai intuitivă noțiunea de arbore, cu precizarea că în informatică "arborii cresc în jos".

Exemplul 3. Considerăm următorul arbore şi modul în care el este aşezat pe niveluri prin alegerea vârfului 5 drept rădăcină.



- *Arborele poate fi considerat un graf orientat*, stabilind pe fiecare muchie sensul de la nivelul superior către nivelul inferior.

Reprezentarea pe niveluri a arborilor face ca noțiunile de *fii* (*descendenți*) ai unui vârf, precum şi de *tată* al unui vârf să aibă semnificații evidente. Un vârf fără descendenți se numeşte *frunză*.

2.3. Arbori binari

Un *arbore binar* este un arbore în care orice vârf are cel mult doi descendenți, cu precizarea că se face distincție între descendentul stâng şi cel drept. Rezultă că un arbore binar nu este propriu-zis un caz particular de arbore.

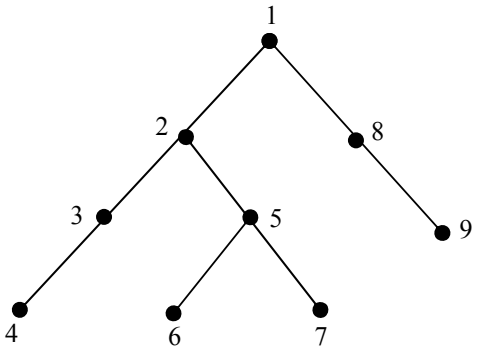
Primele probleme care se pun pentru arborii binari (ca şi pentru arborii oarecare şi pentru grafuri, aşa cum vom vedea mai târziu) sunt:

- modul de reprezentare;
- parcurgerea lor.

Forma standard de reprezentare a unui arbore constă în:
- a preciza rădăcina `rad` a arborelui;
- a preciza pentru fiecare vârf `i` tripletul `st(i)`, `dr(i)` şi `info(i)`, unde acestea sunt respectiv descendentul stâng, descendentul drept şi informaţia ataşată vârfului.

Trebuie stabilită o convenţie pentru lipsa unuia sau a ambilor descendenţi; alegem specificarea lor prin simbolul λ.

Exemplul 4. Considerăm de exemplu următorul arbore binar:



Presupunând că informaţia ataşată fiecărui vârf este chiar numărul său de ordine, avem:
- `rad=1`;
- `st=(2,3,4,λ,6,λ,λ,λ,λ)`;
- `dr=(8,5,λ,λ,7,λ,λ,9,λ)`;
- `info=(1,2,3,4,5,6,7,8,9)`.

Dintre diferitele alte reprezentări posibile, mai menţionăm doar pe cea care se reduce la vectorul său `tata` şi la vectorul `info`. Pentru exemplul de mai sus:
`tata=(λ,1,2,3,2,5,5,1,8)`.

Problema *parcurgerii unui arbore binar* constă în identificarea unei modalităţi prin care, plecând din rădăcină şi mergând pe muchii, să ajungem în toate vârfurile; în plus, atingerea fiecărui vârf este pusă în evidenţă *o singură dată*: spunem că *vizităm* vârful respectiv. Acţiunea întreprinsă la vizitarea unui vârf depinde de problema concretă şi poate fi de exemplu tipărirea informaţiei ataşate vârfului.

Distingem trei modalităţi standard de parcurgere a unui arbore binar:

- **Parcurgerea în preordine**

Se parcurg recursiv în ordine: rădăcina, subarborele stâng, subarborele drept.

Ilustrăm acest mod de parcurgere pentru exemplul de mai sus, figurând îngroşat rădăcinile subarborilor ce trebuie dezvoltați:

1
1, **2**, **8**
1, 2, **3**, **5**, 8, 9
1, 2, 3, 4, 5, 6, 7, 8, 9

Concret, se execută apelul `preord(rad)` pentru procedura:

```
procedure preord(x)
   if x=λ
   then
   else vizit(x); preord(st(x)); preord(dr(x))
end
```

- **Parcurgerea în inordine**

Se parcurg recursiv în ordine: subarborele stâng, rădăcina, subarborele drept.

Ilustrăm acest mod de parcurgere pentru *Exemplul 4*:

1
2, 1, **8**
3, 2, **5**, 1, 8, 9
4, 3, 2, 6, 5, 7, 1, 8, 9

Concret, se execută apelul `inord(rad)` pentru procedura:

```
procedure inord(x)
   if x=λ
   then
   else inord(st(x)); vizit(x); inord(dr(x))
end
```

- **Parcurgerea în postordine**

Se parcurg recursiv în ordine; subarborele stâng, subarborele drept, rădăcina.

Ilustrăm parcurgerea în postordine pentru *Exemplul 4*:

1
2, **8**, 1
3, **5**, 2, 9, 8, 1
4, 3, 6, 7, 5, 2, 9, 8, 1

Concret, se execută apelul `postord(rad)` pentru procedura:

```
procedure postord(x)
   if x=λ
   then
   else postord(st(x)); postord(dr(x)); vizit(x)
end
```

2.4. Aplicații ale arborilor binari

Prezentăm în acest paragraf doi algoritmi de sortare, care folosesc arbori binari.

Fie $a=(a_1,...,a_n)$ vectorul care trebuie sortat (ordonat crescător).

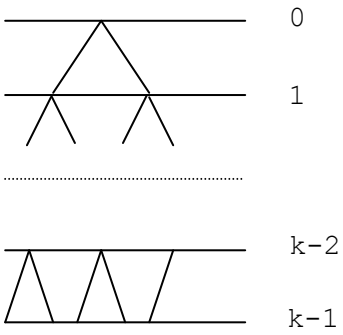
- **Sortarea cu ansamble**

Metoda sortării de ansamble va folosi o *reprezentare implicită a unui vector ca arbore binar*. Acest arbore este construit succesiv astfel:

- rădăcina este 1;
- pentru orice vârf i, descendenții săi stâng şi drept sunt $2i$ şi $2i+1$ (cu condiția ca fiecare dintre aceste valori să nu depăşească pe n). Rezultă că tatăl oricărui vârf i este $tata(i)=\lfloor i/2 \rfloor$.

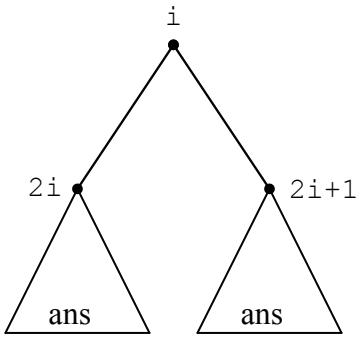
Evident, fiecărui vârf i îi vom ataşa eticheta a_i.

Pentru $2^{k-1} \le n < 2^k$ arborele va avea k niveluri, dintre care numai ultimul poate fi incomplet (pe fiecare nivel $i<k-1$ se află exact 2^i vârfuri).



Vectorul a se numeşte *ansamblu* dacă pentru orice i avem $a_i \ge a_{2i}$ şi $a_i \ge a_{2i+1}$ (dacă fiii există).

Să presupunem că subarborii de rădăcini `2i` şi `2i+1` sunt ansamble. Ne propunem să transformăm arborele de rădăcină `i` într-un ansamblu. Ideea este de a retrograda valoarea a_i până ajunge într-un vârf ai cărui descendenţi au valorile mai mici decât a_i. Acest lucru este realizat de procedura `combin`.

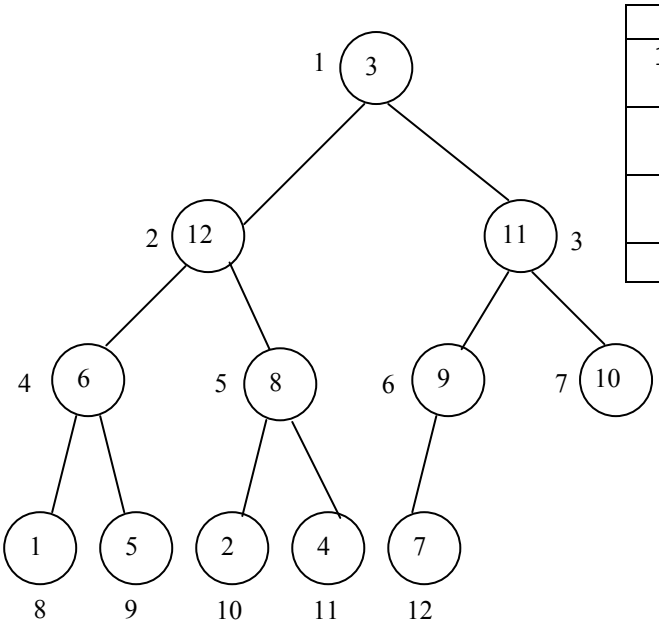



```
procedure combin(i,n)
   j ← 2i; b ← a_i
   while j≤n
      if j<n & a_j<a_{j+1} then j←j+1
      if b>a_j
      then a_⌊j/2⌋ ← b; exit
      else a_⌊j/2⌋ ← a_j; j ←2j
   a_⌊j/2⌋ ← b
end
```

Timpul de executare pentru procedura `combin` este `O(k)=O(log n)`.

Exemplul 5. Pentru:
`n=12, a=(3,12,11,6,8,9,10,1,5,2,4,7)` şi `i=1`, calculele decurg după cum urmează:

n	i	j	b	
12	1	2 4	3	$a_1 \leftarrow 12$
		5 10		$a_2 \leftarrow 8$
		11 22		$a_5 \leftarrow 4$
				$a_{11} \leftarrow 3$



Sortarea vectorului `a` se va face prin apelul succesiv al procedurilor `creare` şi `sortare` prezentate în continuare.

Procedura `creare` transformă vectorul într-un ansamblu; în particular în a_1 se obține cel mai mare element al vectorului.

Procedura `sortare` lucrează astfel:

- pune pe a_1 pe poziția `n` şi reface ansamblul format din primele `n-1` elemente;
- pune pe a_1 pe poziția `n-1` şi reface ansamblul format din primele `n-2` elemente;
- etc.

```
procedure creare
   for i=⌊n/2⌋,1,-1
      combin(i,n)
end
```

```
procedure sortare
   for i=n,2,-1
      a1 ↔ ai; combin(1,i-1)
end
```

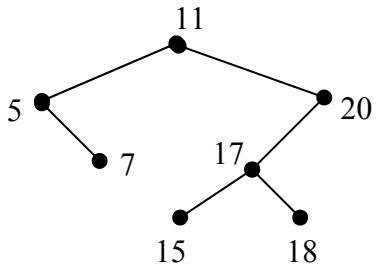
Timpul total de lucru este de ordinul $O(n \cdot \log n)$.

Aşa cum am menționat chiar de la început, structura de arbore este implicită şi este menită doar să clarifice modul de lucru al algoritmului: calculele se referă doar la componentele vectorului.

- **Arbori de sortare**

Un *arbore de sortare* este un arbore binar în care pentru orice vârf informația ataşată vârfului este mai mare decât informațiile vârfurilor din subarborele său stâng şi mai mică decât informațiile vârfurilor din subarborele său drept.

Un exemplu de arbore de sortare este următorul:



Observație. Parcurgerea în inordine a unui arbore de căutare produce informațiile ataşate vârfurilor în ordine crescătoare.

Fie $a=(a_1,...a_n)$ un vector ce trebuie ordonat crescător. Conform observației de mai sus, este suficient să creăm un arbore de sortare în care informațiile vârfului să fie tocmai elementele vectorului. Pentru aceasta este suficient să precizăm modul în care, prin adăugarea unei noi valori, se obține tot un arbore de sortare.

Pentru exemplul considerat:
- adăugarea valorii 6 trebuie să conducă la crearea unui nou vârf, cu informația 6 şi care este descendent stâng al vârfului cu informația 7;
- adăugarea valorii 16 trebuie să conducă la crearea unui nou vârf, cu informația 16 şi care este descendent drept al vârfului cu informația 15.

Presupunem că un vârf din arborele de sortare este o înregistrare sau obiect de tipul `varf`, ce conține câmpurile:
- informația `info` ataşată vârfului;
- descendentul stâng `st` şi descendentul drept `dr` (lipsa acestora este marcată, ca de obicei, prin λ).

Crearea unui nou vârf se face prin apelul funcției `varf_nou`, care întoarce un nou vârf:

```
function varf_nou(info)
    creăm un nou obiect/ o nouă înregistrare x în care informația este info, iar
       descendentul stâng şi cel drept sunt λ;
    return x
end
```

Inserarea unei noi valori `val` (în arborele de rădăcină `rad`) se face prin apelul `adaug(rad,val)`, unde funcția `adaug` întoarce o înregistrare şi are forma:

```
function adaug(x,val)    { se inserează val în subarborele de rădăcină x}
   if x=λ
   then return varf_nou(val)
   else if val<info(x)
        then st(x) ← adaug(st(x),val)
        else dr(x) ← adaug(dr(x),val)
        return x
end
```

Programul principal întreprinde următoarele acțiuni:
- citeşte valorile ce trebuie ordonate şi le inserează în arbore;
- parcurge în inordine arborele de sortare; vizitarea unui vârf constă, de exemplu, în tipărirea informației ataşate.

Prezentăm programul în Java (clasa `IO.java`, folosită pentru citire şi scriere, este descrisă în anexă) :

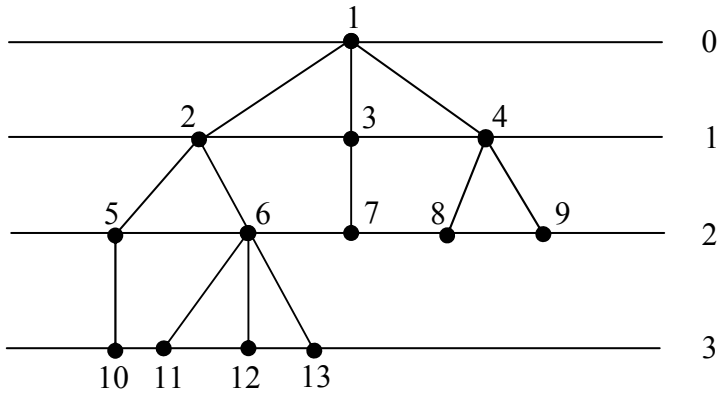
```
class elem {
  int c; elem st,dr;
  elem() { }
  elem(int ch) { c=ch; st=null; dr=null; }
  elem adaug(elem x, int ch) {
    if (x==null) x=new elem(ch);
    else if (ch<x.c) x.st=adaug(x.st,ch);
         else x.dr=adaug(x.dr,ch);
    return x;
  }
  String parcurg(elem x) {
    if (x==null) return("");
    else return( parcurg(x.st) + x.c + " " + parcurg(x.dr));
  }
}
class Arbsort {
  public static void main(String arg[]) {
    elem rad=null; double val;
    elem Ob = new elem(); val = IO.read();
    while ( ! Double.isNaN(val) )
      { rad = Ob.adaug(rad,(int) val); val = IO.read(); }
    IO.writeln(Ob.parcurg(rad));
  }
}
```

2.5. Arbori oarecare

Continuăm cu studiul arborilor oarecare.

Primele probleme care se pun sunt aceleaşi ca pentru arborii binari: modalitățile de reprezentare şi de parcurgere.

Exemplul 6. Considerăm următorul arbore:



Se consideră că arborele este aşezat pe niveluri şi că pentru fiecare vârf există o ordine între descendenții săi.

Modul standard de reprezentare al unui arbore oarecare constă în a memora rădăcina, iar pentru fiecare vârf `i` informațiile:

- `info(i)` = informația ataşată vârfului;
- `fiu(i)` = primul vârf dintre descendenții lui `i`;
- `frate(i)` = acel descendent al tatălui lui `i`, care urmează imediat după `i`.

Ca şi pentru arborii binari, lipsa unei legături este indicată prin λ.

Pentru arborele din *Exemplul 6*:

```
fiu=(2,5,7,8,10,11,λ,λ,λ,λ,λ,λ,λ);
frate=(λ,3,4,λ,6,λ,λ,9,λ,λ,12,13,λ).
```

O altă modalitate de reprezentare constă în a memora pentru fiecare vârf tatăl său. Această modalitate este incomodă pentru parcurgerea arborilor, dar se dovedește utilă în alte situații, care vor fi prezentate în continuare.

În unele cazuri este util să memorăm pentru fiecare vârf atât fiul şi fratele său, cât şi tatăl său.

- **Parcurgerea în preordine**

Se parcurg recursiv în ordine rădăcina şi apoi subarborii care au drept rădăcină descendenții săi. Pentru *Exemplul 6*:

1
1,**2**,**3**,**4**
1,2,**5**,**6**,3,7,4,8,9
1,2,5,10,6,11,12,13,3,7,4,8,9.

Concret, executăm apelul `Apreord(rad)` pentru procedura:

```
procedure Apreord(x)
   if x=λ
   then
   else vizit(x); Apreord(fiu(x)); Apreord(frate(x))
end
```

Ca aplicație, să presupunem că informațiile ataşate vârfurilor sunt funcții de diferite *arități* (aritatea unei funcții este numărul variabilelor de care depinde; o funcție de aritate 0 este o constantă).

Pentru *Exemplul 6*, vectorul de aritate este:
`aritate=(3,2,1,2,1,3,0,0,0,0,0,0,0).`

Rezultatul `1 2 5 10 6 11 12 13 3 7 4 8 9` al parcurgerii în preordine este o formă fără paranteze (dar la fel de consistentă) a scrierii expresiei funcționale:
`1(2(5(10),6(11,12,13)),3(7),4(8,9))`

Această formă se numeşte *forma poloneză directă*.

- **Parcurgerea în postordine**

Se parcurg recursiv în ordine subarborii rădăcinii şi apoi rădăcina. Pentru *Exemplul 5*:

1
2,**3**,**4**,1
5,**6**,2,7,3,8,9,4,1
10,5,11,12,13,6,2,7,3,8,9,4,1.

Concret, executăm apelul `Apostord(rad)` pentru procedura:

```
procedure Apostord(x)
   if x=λ
   then
   else y←fiu(x);
        while y<>λ
           Apostord(y); y←frate(y)
        vizit(x)
end
```

Reluăm aplicația de la parcurgerea în inordine. Dorim să calculăm valoarea expresiei funcționale, cunoscând valorile frunzelor (funcțiilor de aritate 0). Este evident că trebuie să parcurgem arborele în postordine.

- **Parcurgerea pe niveluri**

Se parcurg vârfurile în ordinea distanței lor față de rădăcină, ținând cont de ordinea în care apar descendenții fiecărui vârf. Pentru *Exemplul 5*:

1,2,3,4,5,6,7,8,9,10,11,12,13.

Pentru implementare vom folosi o coadă `C`, în care inițial apare numai rădăcina. Atâta timp cât coada este nevidă, vom extrage primul element, îl vom vizita şi vom introduce în coadă descendenții săi:

```
C ← ∅; C ⇐ rad
while C ≠ ∅
   x ⇐ C; vizit(x);
   y ← fiu(x);
   while y≠λ
      y ⇒ C ; y ← frate(y)
```

Parcurgerea pe niveluri este în general utilă atunci când se caută vârful care este cel mai apropiat de rădăcină şi care are o anumită proprietate/informație.

3 GRAFURI

3.1. Parcurgerea DF a grafurilor neorientate

Fie G=(V,M) un graf neorientat. Ca de obicei, notăm n=|V| şi m=|M|.

Parcurgerea în adâncime a grafurilor (*DF* = *D*epth *F*irst) generalizează parcurgerea în preordine a arborilor oarecare. Eventuala existență a ciclurilor conduce la necesitatea de a marca vârfurile vizitate. Ideea de bază a algoritmului este următoarea: se pleacă dinr-un vârf i_0 oarecare, apelând procedura DF pentru acel vârf. Orice apel de tipul DF(i) prevede următoarele operații:

- marcarea vârfului i ca fiind vizitat;
- pentru toate vârfurile j din lista L_i a vecinilor lui i se execută apelul DF(j) dacă şi numai dacă vârful j nu a fost vizitat.

Simpla marcare a unui vârf ca fiind sau nu vizitat poate fi înlocuită, în perspectiva unor aplicații prezentate în continuare, cu atribuirea unui număr de ordine fiecărui vârf; mai precis, în nrdf(i), inițial egal cu 0, va fi memorat al câtelea este vizitat vârful i; nrdf(i) poartă numele de *numărul (de ordine) DF* al lui i.

Este evident că plecând din vârful i_0 se pot vizita numai vârfurile din componenta conexă a lui i_0. De aceea, în cazul în care graful nu este conex, după parcurgerea componentei conexe a lui i_0 vom repeta apelul DF pentru unul dintre eventualele vârfuri încă neatinse.

```
procedure DF(i)
   ndf ← ndf+1; nrdf(i) ← ndf; vizit(i);
   for toți j∈L_i
      if nrdf(j)=0 then DF(j)
```

Programul principal este:

```
citeşte graful;
ndf←0;
for i=1,n
   nrdf(i) ← 0
for i=1,n
   if nrdf(i)=0 then DF(i)
write(nrdf)
```

Observaţie. Dacă dorim doar să determinăm dacă un graf este conex, vom înlocui al doilea ciclu `for` din programul principal prin:

```
DF(1);
if ndf=n then write('CONEX')
         else write('NECONEX');
```

Observaţie. Parcurgerea DF împarte muchiile grafului în:

- *muchii de avansare*: sunt acele muchii `(i,j)` pentru care în cadrul apelului `DF(i)` are loc apelul `DF(j)`. Aceste muchii formează un graf parțial care este o pădure: fiecare vârf este vizitat exact o dată, deci nu există un ciclu format din muchii de avansare.
- *muchii de întoarcere*: sunt acele muchii ale grafului care nu sunt muchii de avansare.

Determinarea mulțimilor `A` şi `I` a muchiilor de avansare şi întoarcere, precum şi memorarea arborilor parțiali din pădurea DF se poate face astfel:

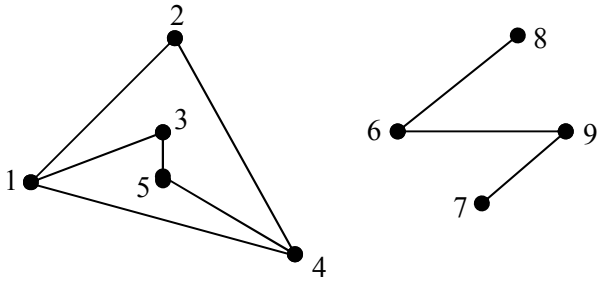
- în programul principal se fac inițializările:

```
A ← ∅; I ← ∅;
for i=1,n
   tata(i) ← 0;
```

- instrucțiunea `if` din procedura `DF` devine:

```
if nrdf(j)=0
then A←A∪{(i,j)}; tata(j) ← i; DF(j)
else if tata(j)≠i
     then I←I∪{(i,j)};
```

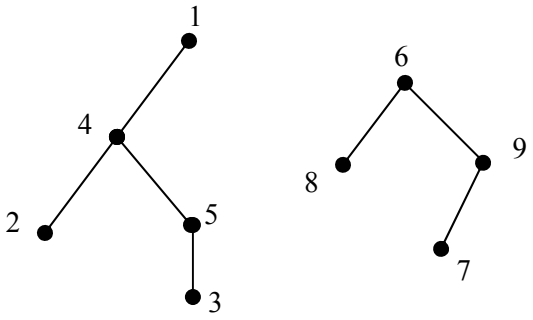
Exemplu. Pentru graful:



cu următoarele liste ale vecinilor vârfurilor:

L_1={4,2,3};	L_2={1,4};	L_3={1,5};
L_4={1,2,5};	L_5={3,4};	L_6={8,9};
L_7={9};	L_8={6}	L_9={6,7}

pădurea DF este formată din următorii arbori parțiali corespunzători componentelor conexe:



Timpul cerut de algoritmul de mai sus este `O(max{n,m})=O(n+m)`, deci algoritmul este liniar, deoarece:

- pentru fiecare vârf `i`, apelul `DF(i)` are loc exact o dată;
- executarea unui apel `DF(i)` necesită un timp proporțional cu `grad(i)=|`L_i`|`; în consecință timpul total va fi proporțional cu `m=|M|`.

Propoziție. Dacă `(i,j)` este muchie de întoarcere, atunci `i` este descendent al lui `j` în arborele parțial ce conține pe `i` şi `j`.

Muchia `(i,j)` este detectată ca fiind muchie de întoarcere în cadrul executării apelului `DF(i)`, deci `nrdf(j)<nrdf(i)`. Deoarece există muchie între vârfurile `i` şi `j`, rezultă că în timpul executării lui `DF(j)` va fi vizitat şi vârful `i`, deci `i` este descendent al lui `j`.

Propoziția de mai sus spune că muchiile de întoarcere leagă totdeauna două vârfuri situate pe aceeaşi ramură a pădurii parțiale DF. În particular, nu există muchii de traversare (care să lege doi descendenți ai aceluiaşi vârf dintr-un arbore DF).

Observație. Un graf este ciclic (conține cel puțin un ciclu) dacă şi numai dacă în timpul parcurgerii sale în adâncime este detectată o muchie de întoarcere.

Aplicație. *Să se determine dacă un graf este ciclic şi în caz afirmativ să se identifice un ciclu.*

Vom memora pădurea formată din muchiile de avansare cu ajutorul vectorului `tata` şi în momentul în care este detectată o muchie de întoarcere `(i,j)` vom lista drumul de la `i` la `j` format din muchii de avansare şi apoi muchia `(j,i)`.

Procedura `DF` va fi modificată astfel:

```
procedure DF(i)
   ndf ← ndf+1; nrdf(i) ← ndf; vizit(i);
   for toți j∈L_i
      if nrdf(j)=0
      then tata(j) ← i; DF(j)
      else if tata(i)≠j
           then k ← i;
                while k≠j
                   write(k,tata(k)); k ← tata(k);
                write(j,i); stop
end.
```

Observații:

- dacă notăm prin `nrdesc(i)` numărul descendenților lui `i` în subarborele de rădăcină `i`, această valoare poate fi calculată plasând după ciclul `for` din procedura `DF` instrucțiunea `nrdesc(i)←ndf-nrdf(i)+1`;
- un vârf `j` este descendent al vârfului `i` în subarborele DF de rădăcină `i` ⇔ `nrdf(i)≤nrdf(j)<nrdf(i)+nrdesc(i)`.

3.2. O aplicație: Problema bârfei

Se consideră `n` persoane. Fiecare dintre ele emite o bârfă care trebuie cunoscută de toate celelalte persoane.

Prin *mesaj* înțelegem o pereche de numere `(i,j)` cu `i,j∈{1,...,n}` şi cu semnificația că persoana `i` transmite persoanei `j` bârfa sa, dar şi toate bârfele care i-au parvenit până la momentul acestui mesaj.

Se cere una dintre cele mai scurte succesiuni de mesaje prin care toate persoanele află toate bârfele.

Cu enunțul de mai sus, o soluție este imediată şi constă în succesiunea de mesaje: `(1,2),(2,3),...,(n-1,n),(n,n-1),(n-1,n-2),...,(2,1)`.

Sunt transmise deci `n-2` mesaje. După cum vom vedea mai jos, acesta este numărul minim de mesaje prin care toate persoanele află toate bârfele.

Problema se complică dacă există persoane care nu comunică între ele (sunt certate) şi deci nu-şi vor putea transmite una alteia mesaje.

Această situație poate fi modelată printr-un graf în care vârfurile corespund persoanelor, iar muchiile leagă persoane care nu sunt certate între ele.

Vom folosi matricea de adiacență a de ordin n în care a_{ij} este 0 dacă persoanele i şi j sunt certate între ele (nu există muchie între i şi j) şi 1 în caz contrar.

Primul pas va consta în detectarea unui arbore parțial; pentru aceasta vom folosi parcurgerea DF. Fiecărei persoane i îi vom ataşa variabila booleană v_i, care este true dacă şi numai dacă vârful corespunzător a fost atins în timpul parcurgerii; inițial toate aceste valori sunt false. Vom pune $a_{ij}=2$ pentru toate muchiile de avansare.

```
v_i ← false, ∀i=1,...n
ndf ← 0; DF(1)
if ndf<n
then write('Problema nu are soluție (graf neconex)')
else rezolvă problema pe arborele parțial obținut
```

unde procedura DF are forma cunoscută:

```
procedure DF(i)
   v_i ← true; ndf ← ndf+1
   for j=1,n
     if a_ij=1 & not v_j
     then a_ij ← 2; DF(j)
end
```

Să observăm că în acest mod am redus problema de la un graf la un arbore! Descriem în continuare modul în care rezolvăm problema pe acest arbore parțial, bineînțeles în ipoteza că problema are soluție (graful este conex).

Printr-o parcurgere în postordine, în care vizitarea unui vârf constă în transmiterea de mesaje de la fiii săi la el, rădăcina (presupusă a fi persoana 1) va ajunge să cunoască toate bârfele. Aceasta se realizează prin apelul postord(1), unde procedura postord are forma:

```
procedure postord(i)
   for j=1,n
      if a_ij=2
      then postord(j); write(j,i)
end
```

În continuare, printr-o parcurgere în preordine a arborelui DF, mesajele vor circula de la rădăcină către frunze. Vizitarea unui vârf constă în transmiterea de mesaje fiilor săi. Pentru aceasta executăm apelul preord(1), unde procedura preord are forma:

```
procedure preord(i)
   for j=1,n
      if a_ij=2
      then write(i,j); preord(j);
end
```

Observăm că atât la parcurgerea în postordine, cât şi la cea în preordine au fost listate n-1 perechi (mesaje), deoarece un arbore cu n vârfuri are n-1 muchii. Rezultă că soluția de mai sus constă într-o succesiune de 2n-2 mesaje. Mai rămâne de demonstrat că acesta este numărul minim posibil de mesaje care rezolvă problema.

Propoziție. Orice soluție pentru problema bârfei conține cel puțin 2n-2 mesaje.

Să considerăm o soluție oarecare pentru problema bârfei.

Punem în evidență *primul* mesaj prin care o persoană a ajuns să cunoască toate bârfele; fie k această persoană. Deoarece celelalte persoane trebuie să le fi emis, înseamnă că până acum au fost transmise cel puțin n-1 mesaje. Dar k este *prima* persoană care a aflat toate bârfele, deci celelalte trebuie să mai afle cel puțin o bârfă. Rezultă că în continuare trebuie să apară încă cel puțin n-1 mesaje. În concluzie, soluția considerată este formată din cel puțin 2n-2 mesaje.

3.3. Circuitele fundamentale ale unui graf

Fie $G=(V,M)$ un graf neorientat conex.

Fie $A=(V,M')$ un arbore parțial al lui G. Muchiile din $M' \setminus M$ sunt muchii de întoarcere, numite şi *corzi*.

Pentru fiecare coardă există un unic drum, format numai din muchii din M', ce uneşte extremitățile corzii. Împreună cu coarda, acest drum formează un ciclu numit *ciclu fundamental*.

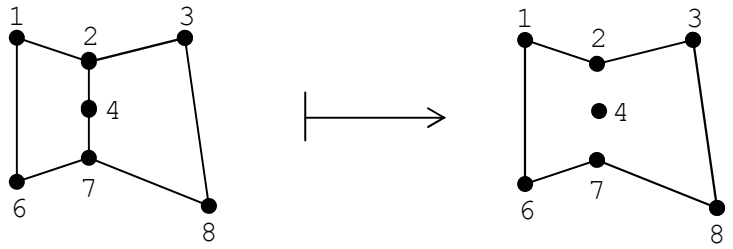
Fie $G_1=(X_1,M_1)$ şi $G_2=(X_2,M_2)$ două grafuri. Definim *suma lor circulară* ca fiind graful:

$$G_1 \oplus G_2 = (X_1 \cup X_2, M_1 \cup M_2 \setminus M_1 \cap M_2)$$

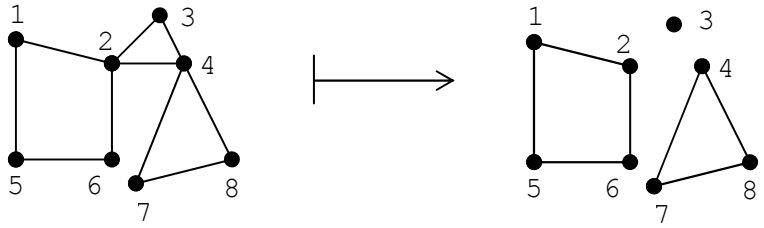
Observații:

1) Operația $\oplus$ este comutativă şi asociativă;
2) Dacă M_1 şi M_2 reprezintă cicluri, atunci $M_1 \oplus M_2$ este tot un ciclu sau o reuniune disjunctă (în privința muchiilor) de cicluri:

Exemple.



$(1,2,4,7,6,1) \oplus (2,3,8,7,4,2) = (1,2,3,8,7,6,1)$

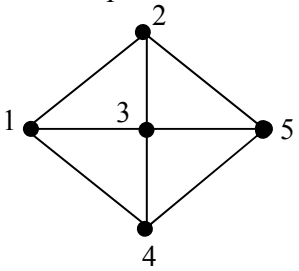


$(1,5,6,2,4,3,2,1) \oplus (7,8,4,3,2,4,7) = (1,5,6,2,1) \cup (7,8,4,7)$

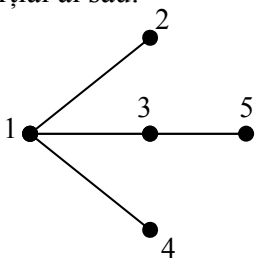
Teoremă. Pentru un graf şi un arbore parțial A al său date, ciclurile fundamentale formează o *bază*, adică sunt îndeplinite condițiile:

1) orice ciclu se poate exprima ca sumă circulară de cicluri fundamentale;
2) nici un ciclu fundamental nu poate fi exprimat ca sumă circulară de cicluri fundamentale.

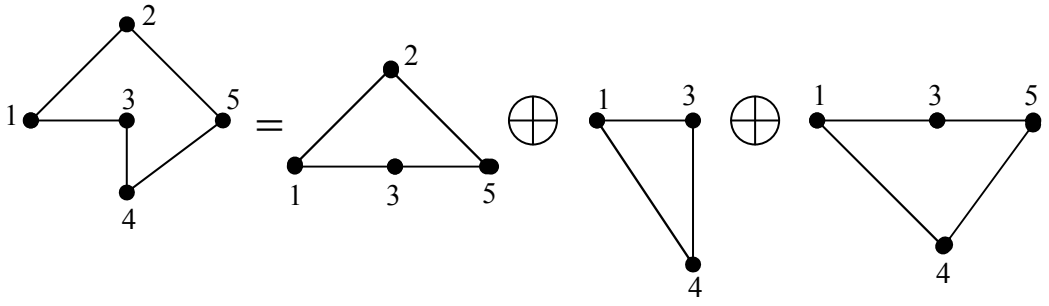
Exemplu. Considerăm următorul graf şi un arbore parțial al său:



Arborele parțial A:



Muchiile de întoarcere sunt (2,3), (3,4), (2,5), (4,5). Ciclul (1,2,5,4,3,1) se poate scrie ca o sumă circulară de cicluri fundamentale astfel:



Demonstrație.

Considerăm un ciclu în G, ale cărui muchii sunt partiționate în $C=\{e_1,...,e_k\}\cup\{e_{k+1},...,e_j\}$ unde $e_1,...,e_k$ sunt corzi, iar $e_{k+1},...,e_j$ sunt muchii din A.

Fie $C(e_1),...,C(e_k)$ ciclurile fundamentale din care fac parte $e_1,...,e_k$. Fie $C'=C(e_1)\oplus...\oplus C(e_k)$. Vom demonstra că $C=C'$ (sunt formate din aceleaşi muchii).

Presupunem că $C\neq C'$. Atunci $C\oplus C'\neq\emptyset$. Să observăm că atât C cât şi C' conțin corzile $e_1,...,e_k$.

Conform unei observații de mai sus, C' şi apoi $C\oplus C'$ sunt cicluri sau reuniuni disjuncte de cicluri. Cum atât C cât şi C' conțin corzile $e_1,...,e_k$ şi în rest muchii din A, rezultă că $C\oplus C'$ conține numai muchii din A, deci nu poate conține un ciclu. Contradicție.

Fie un ciclu fundamental care conține coarda e. Fiind singurul ciclu fundamental ce conține e, el nu se va putea scrie ca sumă circulară de alte cicluri fundamentale.

Consecință. Baza formată din circuitele fundamentale are ordinul $m-n+1$.

Determinarea mulțimii ciclurilor fundamentale

Printr-o parcurgere a arborelui A, putem stabili pentru el legătura $tata$. Atunci pentru orice coardă (i,j) procedăm după cum urmează:

1) Determinăm vectorii:
   $u = (u_1=i, u_2=tata(u_1), ... , u_{nu}=tata(u_{nu-1})=rad)$
   $v = (v_1=j, v_2=tata(v_1), ... , v_{nv}=tata(v_{nv-1})=rad).$
2) Parcurgem simultan vectorii u şi v de la dreapta la stânga şi determinăm cel mai mic indice k cu $u_k=v_k$.
   Atunci ciclul căutat este:
   $u_1=i, u_2, ... , u_k=v_k, v_{k-1}, ... , v_1=j, i$
   Observăm că pentru fiecare coardă, timpul este $O(n)$.

3.4. Componentele biconexe ale unui graf neorientat

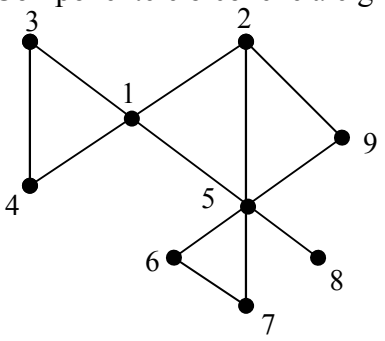
Fie $G=(V,M)$ un graf neorientat, conex.

Un vârf i se numeşte *punct de articulație* dacă prin îndepărtarea sa şi a muchiilor adiacente, graful nu mai rămâne conex.

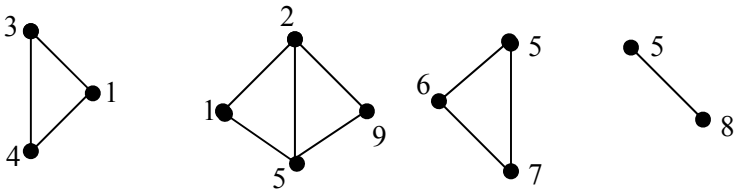
Un graf $G=(V,M)$ se numeşte *biconex* dacă nu are puncte de articulație. Dacă G nu este biconex, se pune în mod natural problema determinării

componentelor sale biconexe, unde prin *componentă biconexă* (sau *bloc*) se înțelege un subgraf biconex maximal.

Exemplu: Componentele biconexe ale grafului:



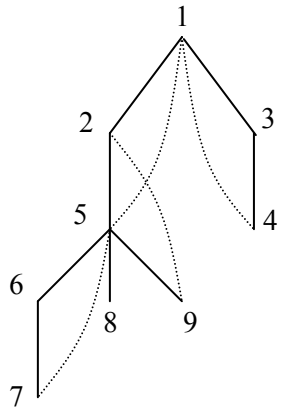
în care elementele din listele vecinilor apar în ordine crescătoare, sunt următoarele:



Să observăm că descompunerea unui graf în componente biconexe determină o partiționare a lui `M`, dar nu a lui `V`.

Prezentăm în continuare un algoritm de complexitate liniară în timp pentru determinarea componentelor biconexe ale unui graf. Algoritmul se bazează pe parcurgerea DF a grafurilor.

Pentru exemplul de mai sus, parcurgerea DF conduce la următorul arbore parțial şi la următoarele numere de ordine DF ataşate vârfurilor (se presupune că în lista vecinilor unui vârf, aceştia apar în ordine crescătoare):



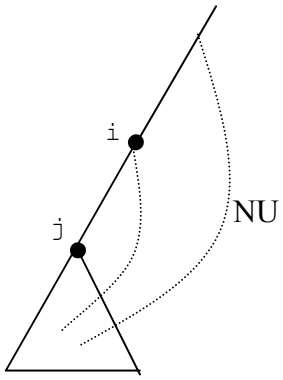
i	nrdf(i)	v(i)
1	1	1
2	2	1
3	8	1
4	9	1
5	3	1
6	4	3
7	5	3
8	6	6
9	7	2

unde muchiile de întoarcere au fost figurate punctat. Punctele de articulație sunt vârfurile 1 şi 5.

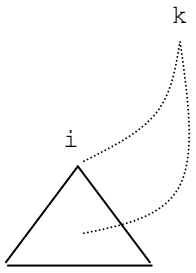
Reamintim faptul că muchiile de întoarcere pot uni doar vârfuri situate pe aceeaşi ramură a arborelui DF, deci nu pot fi muchii de ”traversare”:

Punctele de articulație pot fi caracterizate astfel:

1) rădăcina arborelui DF este punct de articulație dacă şi numai dacă are cel puțin doi descendenți;
2) un vârf `i` diferit de rădăcină este punct de articulație dacă şi numai dacă are un fiu `j` cu proprietatea că nici un vârf din subarborele de rădăcină `j` nu este conectat printr-o muchie de întoarcere cu un predecesor al lui `i`:



Pentru a putea lucra mai uşor cu condiția 2), vom asocia fiecărui vârf `i` o valoare `v(i)` definită astfel:
`v(i)` = *min* { `nrdf(k)` | `k=i` sau `k` legat printr-o muchie de întoarcere la `i` sau la un descendent al lui `i` }.



sau `k=i`

Evident, `v(i)≤nrdf(i)`. Cum orice ciclu elementar este format din muchii de avansare plus exact o muchie de întoarcere, rezultă că `v(i)` este numărul de ordine DF al vârfului `k` cel mai apropiat de rădăcină care se află într-un acelaşi ciclu elementar cu `i`.

Pentru exemplul considerat, valorile lui v apar în tabelul de mai sus.

Condiția 2) se poate reformula acum astfel:

2') un vârf i diferit de rădăcină este punct de articulație dacă şi numai dacă are un fiu j cu v(j)≥ nrdf(i).

Pentru exemplul considerat, v(8)=6 ≥ 3=nrdf(5), deci 5 este punct de articulație.

Definiția lui v poate fi reformulată, astfel încât să fie adecvată parcurgerii DF a grafului:

v(i) = min {α,β,γ} unde:

α = nrdf(i); β = min {v(j)|j fiu al lui i};

γ = min{nrdf(j)|(i,j)∈I},

cu observația că β corespunde cazului când (vezi definiția lui v(i)) k este legat printr-o muchie de întoarcere de un descendent al lui i.

Algoritmul de determinare a componentelor biconexe a unui graf conex foloseşte o stivă S (inițial vidă), în care sunt memorate muchiile componentei biconexe curente:

```
procedure Bloc(i)
   ndf ← ndf+1; nrdf(i) ← ndf; v(i) ← ndf;          { α }
   for toți j∈L_i
       if (i,j) nu a apărut până acum în stiva S
       then (i,j)⇒S
       if nrdf(j)=0 { j devine fiu al lui i în arborele DF }
       then tata(j) ← i; Bloc(j);
            if v(j) ≥ nrdf(i)   { i punct de articulație }
            then repeat
                    α ⇐ S; write (α)
                 until α = (i,j);
            v(i) ← min {v(i),v(j)}                   { β }
       else { (i,j) muchie de întoarcere }
            if j≠tata(i)
            then v(i) ← min {v(i),nrdf(j)}           { γ }
end
```

cu observația că prin apelul Bloc(j) sunt calculate valorile nrdf şi v pentru toate vârfurile din subarborele de rădăcină j în arborele DF.

Programul principal are forma:

```
ndf ← 0; S ⇐ ∅;
```

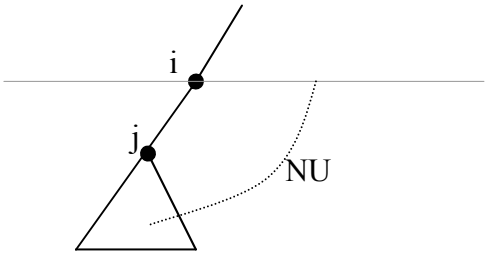
```
for i=1,n
   nrdf(i)←0;
for i=1,n
   if nrdf(i)=0 then Bloc(i);
```

Pentru a demonstra corectitudinea algoritmului este suficient să arătăm că dacă se ajunge de la un vârf `i` la un fiu `j` al său cu `v(j)≥nrdf(i)`, muchiile care apar în `S` începând de la vârf până la şi inclusiv `(i,j)` formează un bloc (o componentă biconexă).

Vom face demonstraţia prin inducţie după numărul `b` de blocuri.

Dacă `b=1`, inegalitatea `v(j)≥nrdf(i)` este satisfăcută doar pentru rădăcina `i` a arborelui DF şi pentru `j` ca unic fiu al său; în momentul verificării acestei condiţii, stiva conţine toate muchiile grafului, iar `(i,j)` se află la baza stivei.

Presupunem afirmaţia (că la fiecare extragere din stivă sunt extrase muchiile unei componente biconexe) adevărată pentru toate grafurile cu mai puţin de `b` componente biconexe şi fie un graf cu `b` blocuri. Fie `i` *primul* vârf pentru care există un fiu `j` cu `v(j)≥nrdf(i)`. Până în acest moment nu a fost scoasă nici o muchie din `S`, iar muchiile din `S` de deasupra lui `(i,j)` sunt muchii incidente cu vârfurile din subarborele de rădăcină `j`, care împreună cu muchia `(i,j)` formează o componentă biconexă:



deoarece din nici un vârf din subarborele de rădăcină `j` nu se ”urcă” prin muchii de întoarcere mai sus de `i`.

După înlăturarea muchiei `(i,j)` şi a celor situate deasupra sa în stivă, algoritmul se comportă ca şi când blocul nu ar fi existat şi deci numărul blocurilor ar fi fost `b-1`. Putem aplica acum ipoteza de inducţie.

Mai observăm că dacă notăm cu i_0 rădăcina arborelui DF, atunci pentru orice fiu `j` al lui i_0 avem `v(j)≥nrdf(`i_0`)` deoarece nici o muchie de întoarcere cu o extremitate în `j` nu poate avea cealaltă extremitate "mai sus" decât i_0. Drept urmare, după ce se coboară pe muchia de avansare `(`i_0`,j)`, se continuă parcurgerea DF şi se revine în i_0, vor fi înlăturate din stivă toate muchiile din

componenta biconexă ce conține (i_0, j). În concluzie şi situația în care rădăcina este punct de articulație este tratată corect.

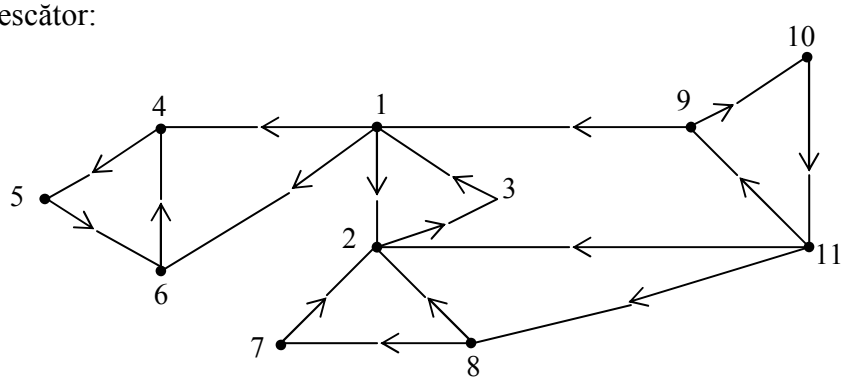
Algoritmul de mai sus pentru determinarea componentelor biconexe este liniar (în $m+n$) deoarece timpul cerut de parcurgerea DF este liniar, iar operațiile cu stiva necesită un timp proporțional cu $m=|M|$.

3.5. Parcurgerea DF a grafurilor orientate

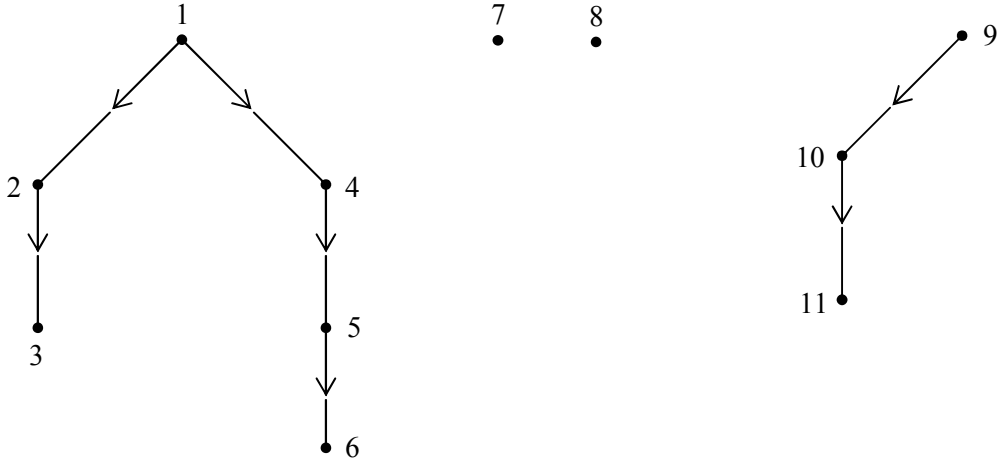
Algoritmul este acelaşi ca la grafuri neorientate.

Arcele de avansare formează o pădure constituită din arbori în care toate arcele au orientarea "de la rădăcină către frunze", numită "pădure DF".

Exemplu. Pentru graful următor, în care listele vecinilor sunt ordonate crescător:



obținem pădurea:



şi vectorul `nrdf = (1,2,3,4,5,6,7,8,9,10,11)`.

Parcurgerea DF împarte arcele `(i,j)` în 3 categorii:

1) *arce de avansare* (pentru ele `nrdf(i)<nrdf(j)`); ele se împart în:
   1.1) arce componente ale pădurii DF;
   1.2) arce ce leagă un vârf de un descendent al său care nu este fiu al său;
2) *arce de întoarcere*, ce leagă un vârf de un predecesor al său în pădurea DF; evident `nrdf(i)>nrdf(j)`;
3) *arce de traversare*: leagă două vârfuri care nu sunt unul descendentul celuilalt.

Pentru exemplul considerat avem:

1.2) : (1,6)
2) : (3,1), (6,4), (11,9)
3) : (7,2), (8,2), (8,7), (9,1), (11,2), (11,8)

Propoziție. Pentru orice arc de traversare `(i,j)` avem `nrdf(i)>nrdf(j)`.

Să presupunem prin absurd că `nrdf(i)<nrdf(j)`. Atunci în momentul în care s-a ajuns prima dată la `i`, vârful `j` nu a fost încă atins. Din modul în care lucrează algoritmul, `(i,j)` va fi arc de avansare; contradicție, deoarece `(i,j)` este arc de traversare.

Observație. Spre deosebire de arcele de traversare, arcele de întoarcere determină un circuit elementar (prin adăugarea unui astfel de arc la pădurea DF ia naştere un circuit elementar). Putem stabili dacă un arc `(i,j)` cu `nrdf(i)>nrdf(j)` este de întoarcere sau de traversare astfel:

```
k←i;
while k≠0 & k≠j
   k←tata(k)
if k=0 then write('traversare')
       else write('întoarcere')
```

3.6. Parcurgerea BF a grafurilor neorientate

Fie un graf `G=(V,M)` şi fie i_0 un vârf al său. În unele situații se pune probleme determinării vârfului `j` cel mai apropiat de i_0 cu o anumită proprietate. Parcurgerea DF nu mai este adecvată.

Parcurgerea pe lățime BF (*B*readth *F*irst) urmărește vizitarea vârfurilor în ordinea crescătoare a distanțelor lor față de i_0. Este generalizată parcurgerea pe

niveluri a arborilor, ținându-se cont că graful poate conține cicluri. Va fi deci folosită o coadă C.

La fel ca şi la parcurgerea DF, vârfurile vizitate vor fi marcate.

Pentru exemplul din primul paragraf al acestui capitol, parcurgerea BF produce vârfurile în următoarea ordine:
1, 4, 2, 3, 5, 6, 8, 9, 7.

Algoritmul următor realizează parcurgea pe lățime a componentei conexe a lui i_0:

```
for i=1,n
   vizitat(i) ← false
C ← ∅; C ⇐ i0; vizitat(i0) ← true
while C ≠ ∅
   i ⇐ C; vizit(i)
      if not vizitat(j)
      then j ⇒ C; vizitat(j)←true
   for toți j vecini ai lui i
```

La fel ca pentru parcurgerea DF, algoritmul poate fi completat pentru parcurgerea întregului graf, pentru determinarea unei păduri în care fiecare arbore este un arbore parțial al unei componente conexe etc.

4 METODA GREEDY

4.1. Descrierea metodei Greedy

Metoda Greedy (*greedy*=lacom) este aplicabilă problemelor de optim.

Considerăm mulțimea finită $A=\{a_1,\ldots,a_n\}$ şi o proprietate p definită pe mulțimea submulțimilor lui A:

$$p: P(A)\to\{0,1\} \text{ cu } \begin{cases} p(\varnothing) = 1 \\ p(X) = 1 \Rightarrow p(Y) = 1,\ \forall Y \subset X \end{cases}$$

O submulțime $S\subset A$ se numeşte *soluție* dacă $p(S)=1$.

Dintre soluții va fi aleasă una care optimizează o funcție de cost $f: P(A)\to\mathbf{R}$ dată.

Metoda urmăreşte evitarea parcurgerii tuturor submulțimilor (ceea ce ar necesita un timp de calcul exponențial), mergându-se "direct" spre soluția optimă. Nu este însă garantată obținerea unei soluții optime; de aceea aplicarea metodei Greedy trebuie însoțită neapărat de o demonstrație.

Distingem două variante generale de aplicare a metodei Greedy:

```
S ← ∅
for i=1,n
   x ← alege(A); A←A\{x}
   if p(S∪{x})=1
   then S←S∪{x}
```

```
prel(A)
S ← ∅
for i=1,n
   if p(S∪{a_i})=1
   then S←S∪{a_i}
```

Prima variantă alege în mod repetat câte un element oarecare al mulțimii A şi îl adaugă soluției curente S numai dacă în acest mod se obține tot o soluție. În a doua variantă procedura `prel` realizează o permutare a elementelor lui A, după care elementele lui A sunt analizate în ordine şi adăugate soluției curente S numai dacă în acest mod se obține tot o soluție.

Observații:

- în algoritmi nu apare funcția f !!
- timpul de calcul este liniar (exceptând prelucrările efectuate de procedura `prel` şi funcția `alege`);

- dificultatea constă în a concepe funcția alege, respectiv procedura prel, în care este "ascunsă" funcția f.

Exemplul 1. Se consideră mulțimea de valori reale $A=\{a_1,...,a_n\}$. Se caută submulțimea a cărei sumă a elementelor este maximă.

Vom parcurge mulțimea şi vom selecta numai elementele pozitive, care vor fi plasate în vectorul soluție s.

```
k←0
for i=1,n
   if a_i>0
   then k←k+1; s_k←a_i
write(s)
```

Exemplul 2. Se cere cel mai lung şir strict crescător cu elemente din vectorul $a=(a_1,...,a_n)$.

Începem prin a ordona crescător elementele vectorului a (corespunzător procedurii prel). Apoi parcurgem vectorul de la stânga la dreapta. Folosim notațiile:
lung = lungimea celui mai lung şir strict crescător;
k = lungimea şirului strict crescător curent;
baza = poziția din a de pe care începe şirul strict crescător curent.

```
k←1; s_1←a_1; lung←1; baza←1
for i=2,n
   if a_i>s_k then k←k+1; s_k ← a_i
             else if k>lung then lung←k; baza←i-k
                  k←1; s_1←a_i
```

De exemplu, dacă în urma ordonării vectorul a este:
a=(1,1,2,3,4,4,5,6,7,8,8), vom obține succesiv:
baza=1; lung=1; s=(1)
baza=2; lung=4; s=(1,2,3,4)
baza=6; lung=5; s=(4,5,6,7,8).

(Contra)exemplul 3. Fie mulțimea $A=\{a_1,...,a_n\}$ cu elemente pozitive. Caut submulțimea de sumă maximă, dar cel mult egală cu M dat.

Dacă procedăm ca în *Exemplul 1*, pentru A=(6,3,4,2) şi M=7 obținem {6}. Dar soluția optimă este {3,4} cu suma egală cu 7.

Continuăm cu prezentarea unor exemple clasice.

4.2. Memorarea textelor pe bandă

Textele cu lungimile L(1),...,L(n) urmează a fi aşezate pe o bandă. Pentru a citi textul de pe poziția k, trebuie citite textele de pe pozițiile 1,2,...,k (conform specificului accesului secvențial pe bandă).

O *soluție* înseamnă o permutare $p \in S_n$.

Pentru o astfel de permutare (ordine de aşezare a textelor pe bandă), timpul necesar pentru a citi textul de pe poziția k este: $T_p(k)=L(p_1)+...+L(p_k)$.

Presupunând textele egal probabile, problema constă în determinarea unei permutări p care minimizează funcția de cost:

$$T(p)=\frac{1}{n}\sum_{k=1}^{n} T_p(k).$$

Să observăm că funcția T se mai poate scrie: $T(p)=\frac{1}{n}\sum_{k=1}^{n}(n-k+1)L(p_k)$

(textul de pe poziția k este citit dacă vrem să citim unul dintre textele de pe pozițiile k,...,n).

Conform strategiei Greedy, începem prin a ordona crescător vectorul L. Rezultă că în continuare L(i)<L(j), $\forall i<j$.

Demonstrăm că în acest mod am obținut modalitatea optimă, adică permutarea identică minimizează funcția de cost T.

Fie $p \in S_n$ optimă, adică p minimizează funcția T. Dacă p este diferită de permutarea identică, atunci $\exists i<j$ cu $L(p_i)>L(p_j)$:

p=(|_____|p_i|_____|p_j|_____|)

Considerăm permutarea p' în care am interschimbat elementele de pe pozițiile i şi j:

p'=(|_____|p_j|_____|p_i|_____|)

Atunci
$$\begin{aligned} n[T(p)-T(p')] &= (n-i+1)L(p_i) + (n-j+1)L(p_j) - \\ &\quad - (n-i+1)L(p_j) - (n-j+1)L(p_i) = \\ &= (j-i)L(p_i)+(i-j)L(p_j) = \\ &= (j-i)[L(p_i)-L(p_j)]>0 \end{aligned}$$

ambii factori fiind pozitivi.

Rezultă că T(p')<T(p). Contradicție.

4.3. Problema continuă a rucsacului

Se consideră un rucsac de capacitate (greutate) maximă G şi n obiecte caracterizate prin:

- greutăţile lor $g_1, \ldots, g_n$;
- câştigurile $c_1, \ldots, c_n$ obţinute la încărcarea lor în totalitate în rucsac.

Din fiecare obiect poate fi încărcată orice fracţiune a sa.

Se cere o modalitate de încărcare de (fracţiuni de) obiecte în rucsac, astfel încât câştigul total să fie maxim.

Prin *soluţie* înţelegem un vector $x=(x_1, \ldots, x_n)$ cu $\begin{cases} x_i \in [0,1], \ \forall i \\ \sum_{i=1}^{n} g_i x_i \le G \end{cases}$

O *soluţie optimă* este o soluţie care maximizează funcţia $f(x)=\sum_{i=1}^{n} c_i x_i$.

Dacă suma greutăţilor obiectelor este mai mică decât G, atunci vom încărca toate obiectele: $x=(1,\ldots,1)$. De aceea presupunem în continuare că $g_1+\ldots+g_n>G$.

Conform strategiei Greedy, ordonăm obiectele descrescător după câştigul la unitatea de greutate, deci lucrăm în ipoteza:

$$\frac{c_1}{g_1} \ge \frac{c_2}{g_2} \ge \ldots \ge \frac{c_n}{g_n} \qquad (*)$$

Algoritmul constă în încărcarea în această ordine a obiectelor, atâta timp cât nu se depăşeşte greutatea G (ultimul obiect pote fi eventual încărcat parţial):

```
G1 ← G { G1 reprezintă greutatea disponibilă }
for i=1,n
   if gi≤G1 then xi←1; G1←G1-gi
            else xi←G1/gi;
                 for j=i+1,n
                   xj ← 0
                 stop
write(x)
```

Am obţinut deci $x=(1,\ldots,1,x_j,0,\ldots,0)$ cu $x_j \in [0,1)$.

Arătăm că soluţia astfel obţinută este optimă.

Fie y soluția optimă: $y=(y_1,\ldots,y_k,\ldots,y_n)$ cu $\begin{cases}\sum_{i=1}^{n} g_i y_i = G \\ \sum_{i=1}^{n} c_i y_i \text{ maxim}\end{cases}$

Dacă $y \neq x$, fie k *prima* poziție pe care $y_k \neq x_k$.

Observații:

- $k \leq j$: pentru $k>j$ se depășește G.
- $y_k < x_k$:
  - pentru $k<j$: evident, deoarece $x_k=1$;
  - pentru $k=j$: dacă $y_k>x_k$ se depășește G.

Considerăm soluția: $y'=(y_1,\ldots,y_{k-1},x_k,\alpha y_{k+1},\ldots,\alpha y_n)$ cu $\alpha<1$ (primele $k-1$ componente coincid cu cele din x). Păstrăm greutatea totală G, deci: $g_k x_k+\alpha(g_{k+1}y_{k+1}+\ldots+g_n y_n)=g_k y_k+g_{k+1}y_{k+1}+\ldots+g_n y_n$. Rezultă:

$$g_k(x_k-y_k)=(1-\alpha)(g_{k+1}y_{k+1}+\ldots+g_n y_n) \qquad (**)$$

Comparăm performanța lui y' cu cea a lui y:

$$\begin{aligned} f(y')-f(y) &= c_k x_k +\alpha c_{k+1}y_{k+1} +\ldots+ \alpha c_n y_n - (c_k y_k+c_{k+1}y_{k+1} +\ldots+c_n y_n) = \\ &= c_k(x_k-y_k) + (\alpha-1)(c_{k+1}y_{k+1}+\ldots+c_n y_n) = \\ &= c_k/g_k[g_k(x_k-y_k)+(\alpha-1)(g_k/c_k c_{k+1}y_{k+1}+\ldots+g_k/c_k c_n y_n)] \end{aligned}$$

Dar $\alpha-1>0$ și $g_k/c_k \leq g_s/c_s$, $\forall s>k$, conform $(*)$. Atunci:

$$f(y')-f(y)>c_k/g_k\ [g_k(x_k-y_k)+(\alpha-1)(g_{k+1}y_{k+1}+\ldots+g_n y_n)]=0$$

conform $(**)$, deci $f(y')>f(y)$. Contradicție.

Problema discretă a rucsacului diferă de cea continuă prin faptul că fiecare obiect poate fi încărcat numai în întregime în rucsac.

Să observăm că aplicarea metodei Greedy eșuează în acest caz. Într-adevăr, aplicarea ei pentru:

$G=5,\ n=3$ și $g=(4,3,2),\ c=(6,4,2.5)$

are ca rezultat încărcarea primul obiect; câștigul obținut este 6. Dar încărcarea ultimelor două obiecte conduce la câștigul superior 6.5.

4.4. Problema arborelui parțial de cost minim

Fie $G=(V,M)$ un graf neorientat cu muchiile etichetate cu costuri strict pozitive. Se cere determinarea unui graf parțial de cost minim.

Ca exemplificare, să considerăm `n` oraşe iniţial nelegate între ele. Pentru fiecare două oraşe se cunoaşte costul conectării lor directe (considerăm acest cost egal cu $+\infty$ dacă nu este posibilă conectarea lor). Constructorul trebuie să conecteze oraşele astfel încât din oricare oraş să se poată ajunge în oricare altul. Ce legături directe trebuie să aleagă constructorul astfel încât costul total al lucrării să fie minim?

Este evident că graful parţial căutat este un arbore (dacă ar exista un ciclu, am putea îndepărta orice muchie din el, cu păstrarea conexităţii şi micşorarea costului total).

Vom aplica metoda Greedy: *adăugăm mereu o muchie de cost minim dintre cele nealese şi care nu formează un ciclu cu precedentele muchii alese*.

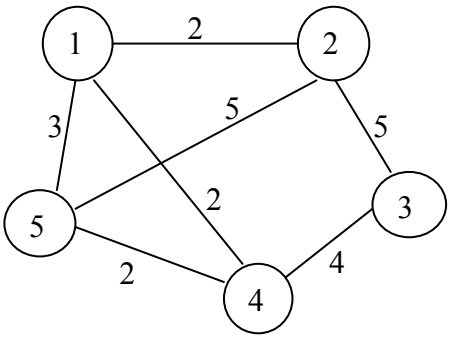
Acest algoritm poartă numele de *algoritmul lui Kruskal*.

Ca de obicei, fie `|V|=n` şi `|M|=m`. Vor fi alese deci `n-1` muchii.

Construim o matrice `mat` cu `m` linii şi trei coloane. Pe fiecare linie apar extremităţile `i` şi `j` ale unei muchii, precum şi costul acestei muchii.

Începem prin a ordona liniile matricii crescător după ultima coloană (a costurilor muchiilor).

Exemplu. Considerăm graful de mai jos şi matricea `mat` ataşată.



1	2	2
1	4	2
4	5	2
1	5	3
3	4	4
2	5	5
2	3	5

Conform algoritmului lui Kruskal, vor fi alese în ordine muchiile:
(1,2), (1,4), (4,5), (3,4)
cu costul total egal cu 10. Muchia (1,5) nu a fost aleasă deoarece formează cu precedentele un ciclu.

Dificultatea principală constă în verificarea faptului că o muchie formează sau nu un ciclu cu precedentele. Plecând de la observaţia că orice soluţie parţială este o pădure, vom asocia fiecărui vârf `i` un reprezentant r_i care identifică componenta conexă (arborele) din care face parte vârful în soluţia parţială. Atunci:

- o muchie (i,j) va forma un ciclu cu precedentele $\Leftrightarrow r_i=r_j$;
- la alegerea (adăugarea) unei muchii (i,j) vom pune $r_k \leftarrow r_j$ pentru orice vârf k cu $r_k=r_i$ (unim doi arbori, deci toate vârfurile noului arbore trebuie să aibă acelaşi reprezentant).

În algoritmul care urmează metoda descrisă, l este numărul liniei curente din matricea mat, nm este numărul de muchii alese, iar cost este costul muchiilor alese.

```
r_i ← i, ∀i=1,n
l ← 1; nm ← 0; cost ← 0
while l≤m & nm<n-1
   i1 ← mat(l,1); i2 ← mat(l,2)
   r1 ← r_i1; r2 ← r_i2
   if r1≠r2
   then nm ← nm+1; cost ← cost+mat(l,3);
        write(i1,i2)
        for k=1,n
           if r_k=r2 then r_k ← r1
   l ← l+1
if nm<n-1 then write('Graf neconex')
          else write('Graf conex. Costul=',cost)
```

Demonstrăm în continuare corectitudinea algoritmului lui Kruskal.

Fie G=(V,M) un graf conex.

P⊂M se numeşte *mulțime promițătoare de muchii* dacă poate fi extinsă la un arbore parțial *P* de cost minim. În particular P nu conține cicluri (este o pădure).

Propoziție. La fiecare pas din algoritmul lui Kruskal muchiile alese formează o mulțime promițătoare P. În plus, muchiile din *P*\P nu au costuri mai mici decât cele din P.

Fie P mulțimea promițătoare a muchiilor selectate la primii k paşi şi fie m muchia considerată la pasul k+1. Deosebim situațiile:

1) dacă m închide un ciclu în P, ea este ignorată. P şi *P* rămân aceleaşi.
2) dacă m nu închide un ciclu în P şi face parte din *P*, noile instanțe ale lui P şi *P* sunt P∪{m} şi *P*.
3) dacă m nu închide un ciclu în P şi nu face parte din *P*, atunci *P*∪{m} are un ciclu. În el există, în afară de m, o muchie m' din *P*\P, deci de cost mai

mare sau egal decât cel al lui m. Fie $P'=P \cup \{m\} \setminus \{m'\}$. P' este tot un arbore parțial de cost minim. Noile instanțe ale lui P şi P sunt $P \cup \{m\}$ şi P'.

În final, o mulțime promițătoare cu n-1 muchii este chiar un arbore parțial de cost minim.

Observație. Tot o ilustrare a metodei Greedy pentru problema enunțată este *algoritmul lui Prim*, care constă în următoarele:

- se începe prin selectarea unui vârf;
- la fiecare pas alegem o muchie (i,j) de lungime minimă cu i selectat, dar j neselectat.

De această dată, la fiecare pas se obține un arbore. Propunem ca exercițiu demonstrarea faptului că după n-1 paşi se obține un arbore parțial de cost minim.

5 METODA BACKTRACKING

Aşa cum s-a subliniat în capitolele anterioare, complexitatea în timp a algoritmilor joacă un rol esențial. În primul rând un algoritm este considerat "acceptabil" numai dacă timpul său de executare este polinomial, adică de ordinul $O(n^k)$ pentru un anumit k; n reprezintă numărul datelor de intrare.

Pentru a ne convinge de acest lucru, vom considera un calculator capabil să efectueze un milion de operații pe secundă. În tabelul următor apar timpii necesari pentru a efectua n^3, 2^n şi 3^n operații, pentru diferite valori mici ale lui n:

	n=20	n=40	n=60
n^3	–	–	0,2 sec
2^n	1 sec	12,7 zile	366 secole
3^n	58 min	3855 secole	10^{13} secole

Chiar dacă în prezent calculatoarele performante sunt capabile să efectueze zeci de miliarde de operații pe secundă, tabelul de mai sus arată că algoritmii exponențiali nu sunt acceptabili.

5.1. Descrierea metodei Backtracking

Fie produsul cartezian $X=X_1 \times \ldots \times X_n$. Căutăm $x \in X$ cu $\varphi(x)=1$, unde $\varphi: X \rightarrow \{0,1\}$ este o proprietate definită pe X.

Din cele de mai sus rezultă că generarea tuturor elementelor produsului cartezian X nu este acceptabilă.

Metoda backtracking încearcă micşorarea timpului de calcul. X este numit *spațiul soluțiilor posibile*, iar φ sintetizează *condițiile interne*.

Vectorul x este construit progresiv, începând cu prima componentă. Nu se trece la atribuirea unei valori lui x_k decât dacă am stabilit valori pentru $x_1,\ldots,x_{k-1}$ şi $\varphi_{k-1}(x_1,\ldots,x_{k-1})=1$. Funcțiile $\varphi_k: X_1 \times \ldots \times X_k \rightarrow \{0,1\}$ se numesc *condiții de continuare* şi sunt de obicei restricțiile lui φ la primele k variabile. Condițiile de continuare sunt strict necesare, ideal fiind să fie şi suficiente.

Distingem următoarele cazuri posibile la alegerea lui x_k:

1) *"Atribuie şi avansează"*: mai sunt valori neconsumate (neanalizate) din X_k şi valoarea x_k aleasă satisface φ_k $\Rightarrow$ se măreşte k.
2) *"Încercare eşuată"*: mai sunt valori neconsumate din X_k şi valoarea x_k aleasă dintre acestea nu satisface φ_k $\Rightarrow$ se va relua, încercându-se alegerea unei noi valori pentru x_k.
3) *"Revenire"*: nu mai există valori neconsumate din X_k (X_k epuizată) $\Rightarrow$ întreaga X_k devine disponibilă şi $k \leftarrow k-1$.
4) *"Revenire după determinarea unei soluţii"*: este reţinută soluţia.

Reţinerea unei soluţii constă în apelarea unei proceduri `retsol` care prelucrează soluţia (o tipăreşte, o compară cu alte soluţii etc.) şi fie opreşte procesul (dacă se doreşte o singură soluţie), fie prevede $k \leftarrow k-1$ (dacă dorim să determinăm toate soluţiile).

Notăm prin $C_k \subset X_k$ mulţimea valorilor consumate din X_k. Algoritmul este următorul:

```
Ci←∅, ∀i;
k←1;
while k>0
   if k=n+1
   then retsol(x); k←k-1;   { revenire după obţinerea unei soluţii }
   else if Ck≠Xk
        then alege v∈Xk\Ck; Ck←Ck∪{v};
             if φk(x1,...,xk-1,v)=1
             then xk←v; k←k+1;     { atribuie şi avansează }
             else                  { încercare eşuată }
        else Ck←∅; k←k-1;          { revenire }
```

Pentru cazul particular $X_1=\ldots=X_n=\{1,\ldots,s\}$, algoritmul se simplifică astfel:

```
k←1; xi←0, ∀i=1,...,n
while k>0
   if k=n+1
   then retsol(x); k←k-1;     { revenire după obţinerea unei soluţii }
   else if xk<s
        then xk←xk+1;
             if φk(x1,...,xk)=1
             then k←k+1;              { atribuie şi avansează }
             else                { încercare eşuată }
        else xk←0; k←k-1;        { revenire }
```

5.2. Exemple

În exemplele care urmează, φ_k va fi notată în continuare prin `cont(k)`. Se aplică algoritmul de mai sus pentru diferite forme ale funcției de continuare.

1) *Colorarea hărților*. Se consideră o hartă. Se cere colorarea ei folosind cel mult `n` culori, astfel încât oricare două țări vecine (cu frontieră comună de lungime strict pozitivă) să fie colorate diferit.

   Fie x_k culoarea curentă cu care este colorată țara `k`.

```
function cont(k: integer): boolean;
   b ← true; i ← 1;
   while b and (i<k)
      if vecin(i,k) & xi=xk
      then b ← false
      else i ← i+1
   cont ← b
end;
```

unde `vecin(i,k)` este `true` dacă şi numai dacă țările `i` şi `k` sunt vecine.

2) *Problema celor n dame*

Se consideră un caroiaj de dimensiuni `n×n`. Prin analogie cu o tablă de şah (`n`=8), se doreşte plasarea a `n` dame pe pătrățelele caroiajului, astfel încât să nu existe două dame una în bătaia celeilalte (adică să nu existe două dame pe aceeaşi linie, coloană sau diagonală).

Evident, pe fiecare linie vom plasa exact o damă. Fie x_k coloana pe care este plasată dama de pe linia `k`.

Damele de pe liniile `i` şi `k` sunt:

- pe aceeaşi coloană: dacă $x_i=x_k$;
- pe aceeaşi diagonală: dacă $|x_i-x_k|=k-i$.

```
function cont(k:integer): boolean;
   b ← true; i ← 1;
   while b and i<k
      if |xi-xk|=k-i or xi=xk
      then b ← false
      else i ← i+1;
   cont ← b
end;
```

3) Problema ciclului hamiltonian

Se consideră un graf neorientat. Un ciclu hamiltonian este un ciclu care trece exact o dată prin fiecare vârf al grafului.

Pentru orice ciclu hamiltonian putem presupune că el pleacă din vârful 1. Vom nota prin x_i al i-lea vârf din ciclu.

Un vector $x=(x_1,...,x_n)$ este *soluție* dacă:

1) $x_1=1$ şi
2) $\{x_2,...,x_n\}=\{2,...,n\}$ şi
3) x_i, x_{i+1} vecine, $\forall i=1,...,n-1$ şi
4) x_n, x_1 vecine.

Vom considera că graful este dat prin matricea sa de adiacență.

```
function cont(k:integer):boolean;
   if a(x_{k-1},x_k)=0
   then cont ← false
   else i ← 1; b ← true;
        while b & (i<k)
           if x_k=x_i then b ← false
                      else i ← i+1
        if k=n then b ← b ∧ a(x_n,x_1)=1
        cont ← b
end
```

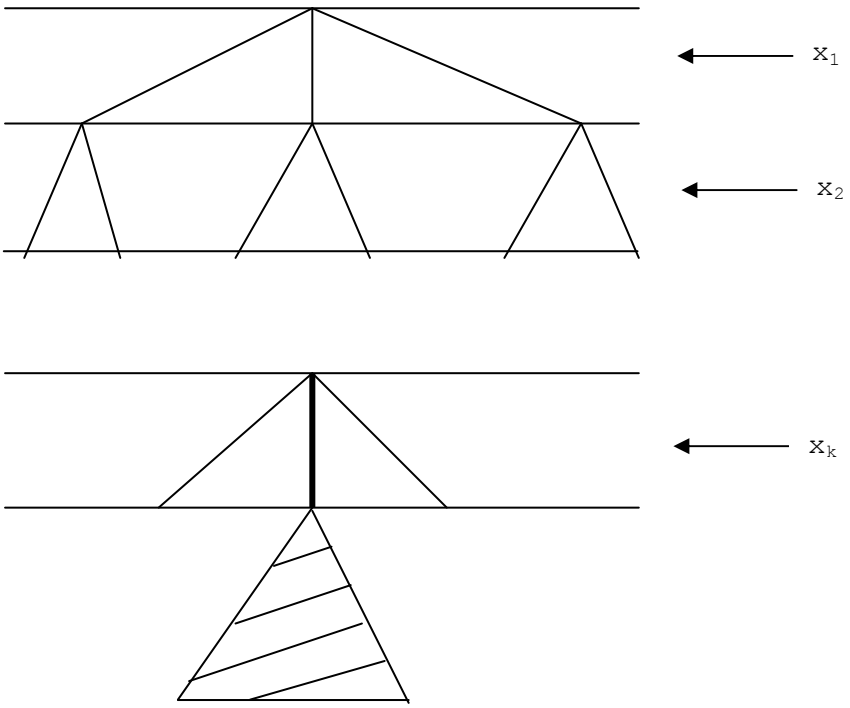
5.3. Esența metodei Backtracking

Metoda backtracking poate fi descrisă astfel:

Backtracking = parcurgerea limitată *) în adâncime a unui arbore

*) conform condițiilor de continuare

Rolul condițiilor de continuare este ilustrat în figura ce urmează. Dacă pentru x_k este aleasă o valoare ce nu satisface condițiile de continuare, atunci la parcurgerea în adâncime este evitată parcurgerea unui întreg subarbore.



5.4. Variante

Variantele cele mai uzuale întâlnite în aplicarea metodei backtracking sunt următoarele:

- soluțiile pot avea un număr variabil de componente *şi/sau*
- dintre soluții alegem una care optimizează o funcție dată.

Exemplu. Fie şirul $a=(a_1,\ldots,a_n)\in\mathbf{Z}^n$. Căutăm un subşir strict crescător de lungime maximă.

Căutăm deci indicii $x_1,\ldots,x_k$ care satisfac condițiile:

1) $1\le x_1<\ldots<x_k\le n$

2) $a_{x_1}<a_{x_2}<\ldots<a_{x_k}$

3) k maxim.

Pentru $n=8$ şi $a=(1,4,2,3,7,5,8,6)$ va rezulta $k=5$.

În această problemă vom înțelege prin *soluție posibilă* o soluție care nu poate fi continuată, ca de exemplu $(4,7,8)$.

Fie `xf` şi `kf` soluția optimă curentă şi lungimea sa. Procedăm astfel:

- Completăm la capetele şirului cu $-\infty$ şi $+\infty$:

$a_0 \leftarrow -\infty$; n←n+1; $a_n \leftarrow +\infty$;

- Funcția `cont` are următoarea formă:

```
function cont(k)
  cont ← a[x_{k-1}] < a[x_k]
end;
```

- Procedura `retsol` are forma:

```
procedure retsol(k)
   if k>kf then xf←x; kf←k;
end;
```

- Algoritmul backtracking se modifică astfel:

```
k←1; x0←0; x1←0; kf←0;
while k>0
   if xk<n
   then xk←xk+1;
        if cont(k)
        then if xk=n                { an=+∞ }
             then retsol(k); k←k-1
             else k←k+1; xk←xk-1
        else
   else k←k-1;
```

Observație. Se face tot o parcurgere limitată în adâncime a unui arbore.

5.5. Abordarea recursivă

Descriem abordarea recursivă pentru $X_1=...=X_n=\{1,...,s\}$.
Apelul inițial este: `back(1)`.

```
procedure back(k)
   if k=n+1
   then retsol
   else for i=1,s
           xk←i;
           if cont(k) then back(k+1);
                              revenirea din recursivitate
end
```

Exemplu. Dorim să producem toate şirurile de n paranteze ce se închid corect.

Este evident că problema are soluţii dacă şi numai dacă n este par.

Fie $nr_($ = numărul de paranteze deschise până la poziţia curentă şi $nr_)$ = numărul de paranteze deschise până la poziţia curentă. Fie $dif = nr_(-nr_)$. Atunci trebuie îndeplinite condiţiile:

$$\begin{cases} dif \geq 0 & \text{pentru } k<n; \\ dif = 0 & \text{pentru } k=n. \end{cases}$$

Pornirea algoritmului backtracking se face prin:

$a_1 \leftarrow$'('; dif$\leftarrow$1; back(2);

Procedura back are următoarea formă:

```
procedure back(k)
   if k=n+1
   then retsol          {scrie soluţia}
   else ak←'( '; dif++;
              if dif ≤ n-k then back(k+1)
              dif--;
        ak←')'; dif--;
              if dif≥0 then back(k+1)
              dif++;
end.
```

Observaţie. În exemplul tratat backtracking-ul este *optimal*, deoarece se avansează dacă şi numai dacă există şanse de obţinere a unei soluţii. Cu alte cuvinte, condiţiile de continuare nu sunt numai necesare, dar şi suficiente.

5.6. Metoda backtracking în plan

Se consideră un caroiaj (matrice) A cu m linii şi n coloane. Poziţiile pot fi:

- libere: $a_{ij}=0$;
- ocupate: $a_{ij}=1$.

Se mai dă o poziţie (i_0, j_0). Se caută toate drumurile care ies în afara matricii, trecând numai prin poziţii libere.

Variante:

- cum putem ajunge într-o poziţie (i_1, j_1) dată?
- se cere determinarea componentelor conexe.

Procedăm astfel:

- Mişcările posibile sunt date printr-o matrice `depl` cu două linii şi `ndepl` coloane. De exemplu, dacă deplasările permise sunt cele către poziţiile vecine situate la Est, Nord, Vest şi Sud, matricea are forma:
  $\texttt{depl} = \begin{pmatrix} 1 & 0 & -1 & 0 \\ 0 & -1 & 0 & 1 \end{pmatrix}$
- Bordăm matricea cu 2 pentru a nu studia separat ieşirea din matrice; în acest mod s-au introdus linia `0` şi linia `m+1`, precum şi coloanele `0` şi `n+1`.
- Pentru refacerea drumurilor, pentru fiecare poziţie atinsă memorăm legătura la poziţia precedentă.
- Dacă poziţia e liberă şi putem continua, punem a_{ij}=-1 (a fost atinsă), continuăm şi apoi repunem $a_{ij} \leftarrow 0$ (întoarcerea din recursivitate).

Programul în Java are următoarea formă (reamintim că prezentarea clasei `IO.java` este făcută în anexă):

```
class elem {
  int i,j; elem prec;
  static int m,n,i0,j0,ndepl;
  static int[][] mat;
  static int[][] depl = { {1,0,-1,0}, {0,-1,0,1} };
  static { ndepl = depl[0].length; }

  elem() {
      int i,j;
      IO.write("m,n = "); m = (int) IO.read();
      n = (int) IO.read();           //m+2,n+2
      mat = new int[m][n];
      for(i=1; i<m-1; i++)
        for(j=1; j<n-1; j++) mat[i][j] = (int) IO.read();
      for (i=0;i<n;i++) {mat[0][i] = 2; mat[m-1][i] = 2;}
      for (j=0;j<m;j++) {mat[j][0] = 2; mat[j][n-1] = 2;}
      IO.write("i0,j0 = "); i0 = (int) IO.read();
      j0 = (int) IO.read();
  }

  elem(int ii,int jj,elem x) { i=ii; j=jj; prec=x; }

  String print(elem x) {
      if (x == null) return "(" + i + "," + j + ")";
      else return x.print(x.prec)+" "+"("+i+","+j+")";
  }
```

```
  void p() {
       elem x; int ii,jj;
       for (int k=0; k<ndepl; k++) {
         ii = i+depl[0][k]; jj = j+depl[1][k];
         if (mat[ii][jj] == 1);
         else if (mat[ii][jj]==2) IO.writeln(print(prec));
         else if (mat[ii][jj]==0) {
              mat[i][j] = -1; x = new elem(ii,jj,this);
              x.p(); mat[i][j] = 0;
         }
       }
  }
}
class DrumPlan {
  public static void main(String[] args) {
    new elem();
    elem start = new elem(elem.i0,elem.j0,null);
    start.p();
  }
}
```

6 METODA DIVIDE ET IMPERA

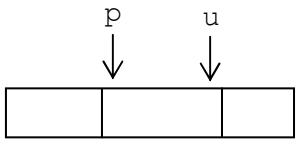
Metoda *Divide et Impera* ("desparte şi stăpâneşte") constă în împărţirea repetată a unei probleme de dimensiuni mari în mai multe subprobleme de acelaşi tip, urmată de rezolvarea acestora şi combinarea rezultatelor obţinute pentru a determina rezultatul corespunzător problemei iniţiale. Pentru fiecare subproblemă procedăm în acelaşi mod, cu excepţia cazului în care dimensiunea ei este suficient de mică pentru a fi rezolvată direct.

Este evident caracterul recursiv al acestei metode.

6.1. Schema generală

Descriem schema generală pentru cazul în care aplicăm metoda pentru o prelucrare oarecare asupra elementelor unui vector. Funcţia `DivImp`, care întoarce rezultatul prelucrării asupra unei subsecvenţe $a_p,...,a_u$, va fi apelată prin `DivImp(1,n)`.

```
function DivImp(p,u)
   if u-p<ε
   then r ← Prel(p,u)
   else m ← Interm (p,u);
        r1 ← DivImp(p,m);
        r2 ← DivImp(m+1,u);
        r ← Combin(r1,r2)
   return r
end;
```



unde:

- funcţia `Interm` întoarce un indice în intervalul `p..u`; de obicei `m=⌊(p+u)/2⌋`;
- funcţia `Prel` este capabilă să întoarcă rezultatul subsecvenţei `p..u`, dacă aceasta este suficient de mică;
- funcţia `Combin` întoarce rezultatul asamblării rezultatelor parţiale `r1` şi `r2`.

Exemple:

- Maximul elementelor unui vector poate fi evident calculat folosind metoda Divide et Impera;
- Parcurgerile în preordine, inordine şi postordine ale unui arbore binar, precum şi sortarea folosind arbori de sortare, urmează întocmai această metodă.

6.2. Căutarea binară

Se consideră vectorul $a=(a_1, \ldots, a_n)$ ordonat crescător şi o valoare x. Se cere să se determine dacă x apare printre componentele vectorului.

Problema enunțată constituie un exemplu pentru cazul în care problema se reduce la o singură subproblemă, deci dispare pasul de recombinare a rezultatelor subproblemelor.

Ținând cont de faptul că a este ordonat crescător, vom compara pe x cu elementul din "mijlocul" vectorului. Dacă avem egalitate, algoritmul se încheie; în caz contrar vom lucra fie pe "jumătatea" din stânga, fie pe cea din dreapta.

Vom adăuga $a_0=-\infty$, $a_{n+1}=+\infty$. Căutăm perechea (b,i) dată de:

- $(true,i)$ dacă $a_i=x$;
- $(false,i)$ dacă $a_{i-1}<x<a_i$.

Deoarece problema se reduce la o singură subproblemă, nu mai este necesar să folosim recursivitatea.

Algoritmul este următorul:

```
procedure CautBin
   p ← 1; u ← n
   while p≤u
      i ← ⌊(p+u)/2⌋
      case ai>x : u ← i-1
           ai=x : write(true,i); stop
           ai<x : p ← i+1
   write(false,p)
end
```

Algoritmul necesită o mică analiză, legată de corectitudinea sa parțială. Mai precis, ne întrebăm: când se ajunge la $p>u$?

- pentru cel puțin 3 elemente : nu se poate ajunge la $p>u$;

- pentru 2 elemente, adică pentru u=p+1: se alege i=p. Dacă $x<a_i$, atunci u←p-1. Se observă că se iese din ciclul while şi $a_{i-1}<x<a_i=a_p$;
- pentru un element, adică p=u: se alege i=p=u. Dacă $x<a_i$ atunci u←p-1, iar dacă $x>a_i$ atunci p←u+1; în ambele cazuri se părăseşte ciclul while şi se tipăreşte un rezultat corect.

6.3. Problema turnurilor din Hanoi

Se consideră 3 tije. Inițial, pe tija 1 se află n discuri cu diametrele decrescătoare privind de la bază către vârf, iar pe tijele 2 şi 3 nu se află nici un disc. Se cere să se mute aceste discuri pe tija 2, ajutându-ne şi de tija 3. Trebuie respectată condiția ca în permanență, pe orice tijă, sub orice disc să se afle baza tijei sau un disc de diametru mai mare.

O *mutare* este notată prin (i,j) şi semnifică deplasarea discului din vârful tijei i deasupra discurilor aflate pe tija j. Se presupune că mutarea este corectă (vezi condiția de mai sus).

Fie *H*(m;i,j) şirul de mutări prin care cele m discuri din vârful tijei i sunt mutate peste cele de pe tija j, folosind şi a treia tijă, al cărei număr este evident 6-i-j. Problema constă în a determina *H*(n;1,2).

Se observă că este satisfăcută relația:

H(m;i,j)= *H*(m-1;i,6-i-j) (i,j) *H*(m-1;6-i-j,j) (*)

cu respectarea condiției din enunț. Deci problema pentru m discuri a fost redusă la două probleme pentru m-1 discuri, al căror rezultat este asamblat conform (*).

Corespunzător, vom executa apelul Hanoi(n,1,2), unde procedura Hanoi are forma:

```
procedure Hanoi(n,i,j)
   if n=1
   then write(i,j)
   else k←6-i-j;
        Hanoi(n-1,i,k); Hanoi(1,i,j); Hanoi(n-1,k,j)
end
```

Observație. Numărul de mutări este 2^n-1.

6.4. Sortarea prin interclasare

Fie $a=(a_1,\ldots,a_n)$ vectorul care trebuie ordonat crescător.

Ideea este următoarea: împărțim vectorul în doi subvectori, ordonăm crescător fiecare subvector şi asamblăm rezultatele prin *interclasare*. Se aplică deci întocmai metoda Divide et Impera.

Începem cu procedura de interclasare. Fie secvența de indici `p..u` şi fie `m` un indice intermediar. Presupunând că $(a_p,...,a_m)$ şi $(a_{m+1},...,a_u)$ sunt ordonați crescător, procedura `Inter` va ordona crescător întreaga secvență $(a_p,...,a_u)$.

Mai precis, vom folosi notațiile:
`k1` = indicele curent din prima secvență;
`k2` = indicele curent din a doua secvență;
`k3` = poziția pe care va fi plasat cel mai mic dintre a_{k1} şi a_{k2} în vectorul auxiliar `b`.

```
procedure Inter(p,m,u)
   k1←p; k2←m+1; k3←p;
   while k1≤m & k2≤u
      if ak1<ak2 then bk3←ak1; k1←k1+1
                 else bk3←ak2; k2←k2+1
      k3←k3+1
   if k1>m       { au fost epuizate elementele primei subsecvențe }
   then for i=k2,u
           bk3←ai; k3←k3+1
   else for i=k1,m
           bk3←ai; k3←k3+1
   for i=p,u
      ai←bi
end
```

Timpul de calcul este de ordinul `O(u-p)`, adică liniar în lungimea secvenței analizate.

Programul principal urmează întocmai strategia Divide et Impera, deci se face apelul `SortInter(1,n)`, unde procedura recursivă `SortInter` are forma:

```
procedure SortInter(p,u)
   if p=u
   then
   else m ← ⌊(p+u)/2⌋;
        SortInter(p,m); SortInter(m+1,u);
        Inter(p,m,u)
end
```

Calculăm în continuare timpul de executare $T(n)$, unde $T(n)$ se poate scrie:

- t_0 (constant), pentru $n=1$;
- $2T(n/2)+an$, pentru $n>1$, unde a este o constantă: problema de dimensiune n s-a descompus în două subprobleme de dimensiune $n/2$, iar combinarea rezultatelor s-a făcut în timp liniar (prin interclasare).

Presupunem că $n=2^k$. Atunci:

$$
\begin{aligned}
T(n) = T(2^k) &= 2\,T(2^{k-1}) + a\,2^k = \\
&= 2[2T(2^{k-2}) + a\,2^{k-1}] + a\,2^k = 2^2T(2^{k-2}) + 2\,a\,2^k = \\
&= 2^2[T(2^{k-3}) + a\,2^{k-2}] + 2\,a\,2^k = 2^3T(2^{k-3}) + 3\,a\,2^k = \\
&\ldots \\
&= 2^iT(2^{k-i}) + i\cdot a\cdot 2^k = \\
&\ldots \\
&= 2^kT(0) + k\,a\,2^k = nt_0 + a\cdot n\cdot \log_2 n.
\end{aligned}
$$

Rezultă că $T(n)=0(n\cdot \log n)$.

Se observă că s-a obținut același timp ca și pentru sortarea cu ansamble.

Mențiune. Se poate demonstra că acest timp este optim.

6.5. Metoda Quicksort

Prezentăm încă o metodă de sortare a unui vector $a=(a_1,\ldots,a_n)$. Va fi aplicată tot metoda Divide et Impera. Şi de această dată fiecare problemă va fi descompusă în două subprobleme mai mici de aceeaşi natură, dar nu va mai fi necesară combinarea (asamblarea) rezultatelor rezolvării subproblemelor.

Fie $(a_p,\ldots,a_u)$ secvența curentă care trebuie sortată. Vom *poziționa* pe a_p în secvența $(a_p,\ldots,a_u)$, adică printr-o permutare a elementelor secvenței $x=a_p$ va trece pe o poziție k astfel încât:

- toate elementele aflate la stânga poziției k vor fi mai mici decât x;
- toate elementele aflate la dreapta poziției k vor fi mai mari decât x.

În acest mod a_p va apărea pe poziția sa finală, rămânând apoi să ordonăm crescător elementele aflate la stânga sa, precum şi pe cele aflate la dreapta sa.

Fie poz funcția cu parametrii p şi u care întoarce indicele k pe care va fi poziționat a_p în cadrul secvenței $(a_p,\ldots,a_u)$.

Atunci sortarea se realizează prin apelul $QuickSort(1,n)$, unde procedura $QuickSort$ are forma:

```
procedure QuickSort(p,u)
   if p≥u
   then
   else k ← poz(p,u); QuickSort(p,k-1); QuickSort(k+1,u)
end
```

Funcția `poz` lucrează astfel:

```
function poz(p,u)
   i←p; j←u; ii←0; jj←-1
   while i<j
      if ai<aj
      then
      else ai ↔ aj; (ii,jj) ← (-ii,-jj)
      i←i+ii; j←j+jj          (*)
   poz ← i
end
```

Să urmărim cum decurg calculele pentru secvența:
$(a_4,\ldots,a_{11})=(6,3,2,5,8,1,9,7)$

- se compară 6 cu $a_{11},a_{10},\ldots$ până când găsim un element mai mic. Acesta este $a_9=1$. Se interschimbă 6 cu 1. Acum secvența este $(1,3,2,5,8,6,9,7)$ şi vom lucra în continuare pe subsecvența $(3,2,5,8,6)$, schimbând direcția de comparare conform $(*)$;
- 6 va fi comparat succesiv cu $3,2,\ldots$ până când găsim un element mai mare. Acesta este $a_8=8$. Se interschimbă 6 cu 8.

Se obține astfel $(1,3,2,5,6,8,9,7)$, în care la stânga lui 6 apar valori mai mici, iar la dreapta lui 6 apar valori mai mari, deci l-am poziționat pe 6 pe poziția 8, valoare întoarsă de funcția `poz`.

Observație. Cazul cel mai defavorabil pentru metoda Quicksort este cel în care vectorul este deja ordonat crescător: se compară a_1 cu $a_2,\ldots,a_n$ rezultând că el se află pe poziția finală, apoi se compară a_2 cu $a_3,\ldots,a_n$ rezultând că el se află pe poziția finală etc. Timpul în acest caz este de ordinul $O(n^2)$.

Trecem la calculul timpului *mediu* de executare al algoritmului Quicksort. Vom număra câte comparări se efectuează (componentele vectorului nu sunt neapărat numere, ci elemente dintr-o mulțime ordonată oarecare). Timpul mediu este dat de formulele:

$$\begin{cases} T(n) = n - 1 + \dfrac{1}{n}\sum_{k=1}^{n}[T(k-1)+T(n-k)] \\ T(1) = T(0) = 0 \end{cases}$$

deoarece:

- în cazul cel mai defavorabil a_1 se compară cu celelalte n-1 elemente;
- a_1 poate fi poziționat pe oricare dintre pozițiile k=1,2,...,n; considerăm aceste cazuri echiprobabile;
- T(k-1) este timpul (numărul de comparări) necesar ordonării elementelor aflate la stânga poziției k, iar T(n-k) este timpul necesar ordonării elementelor aflate la dreapta poziției k.

$$nT(n) = n(n-1)+2[T(0)+T(1)+...+T(n-1)]$$
$$(n-1)T(n-1) = (n-1)(n-2)+2[T(0)+...+T(n-2)]$$

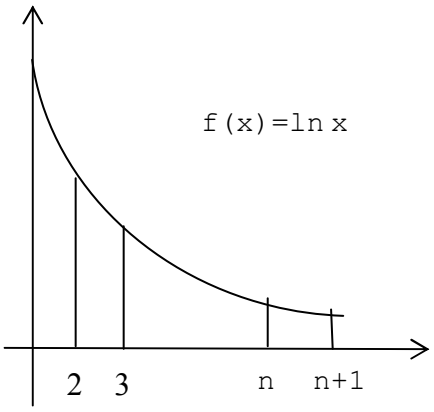
Scăzând cele două relații obținem:

$nT(n)-(n-1)T(n-1) = 2(n-1)+ 2T(n-1)$, deci:

$nT(n) = (n+1)T(n-1)+2(n-1)$.

Împărțim cu n(n+1):

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2(n-1)}{n(n+1)}$$
$$\frac{T(n)}{n+1} = \frac{T(n-1)}{T(n)} + 2\left(\frac{2}{n+1} - \frac{1}{n}\right)$$
$$\frac{T(n-1)}{n} = \frac{T(n-2)}{T(n-1)} + 2\left(\frac{2}{n} - \frac{1}{n-1}\right)$$
$$\cdots\cdots\cdots\cdots\cdots\cdots\cdots$$
$$\frac{T(2)}{3} = \frac{T(1)}{2} + 2\left(\frac{2}{3} - \frac{1}{2}\right)$$



Prin adunarea relațiilor de mai sus, obținem:

$$\frac{T(n)}{n+1} = 2\left(\frac{1}{n+1} + \frac{1}{n} + \ldots + \frac{1}{3}\right) + \frac{2}{n+1} - 1$$

Cum suma ultimilor doi termeni este negativă, rezultă:

$$\frac{T(n)}{n+1} \le 2\int_2^{n+1} \frac{1}{x}dx = 2\ln x|_2^{n+1} \le 2\ln(n+1)$$

(am folosit o inegalitate bazată pe sumele Rieman pentru funcția f(x)=ln x).

Deci $T(n)=0(n\cdot\log n)$.

Încheiem cu mențiunea că metoda Divide et Impera are o largă aplicativitate şi în calculul paralel.

7 METODA PROGRAMĂRII DINAMICE

Vom începe prin a enunța o problemă generală şi a trece în revistă mai mulți algoritmi de rezolvare. Abia după aceea vom descrie metoda programării dinamice.

7.1. O problemă generală

Fie A şi B două mulțimi oarecare.

Fiecărui element $x \in A$ urmează să i se asocieze o valoare $v(x) \in B$.

Inițial v este cunoscută doar pe submulțimea $X \subset A$, $X \neq \emptyset$.

Pentru fiecare $x \in A \setminus X$ sunt cunoscute:

$A_x \subset A$: mulțimea elementelor din A de a căror valoare depinde $v(x)$;

f_x : funcție care specifică dependența de mai sus. Dacă $A_x=\{a_1,\ldots,a_k\}$, atunci $v(x)=f_x(v(a_1),\ldots,v(a_k))$.

Se mai dă $z \in A$.

Se cere să se calculeze, dacă este posibil, valoarea $v(z)$.

Exemplu.

$A=\{1,2,\ldots,13\}$; $X=\{1,2,6,7,8,9,10\}$;

$A_3=\{1,2\}$; $A_4=\{1,2,3\}$; $A_5=\{1,4\}$;

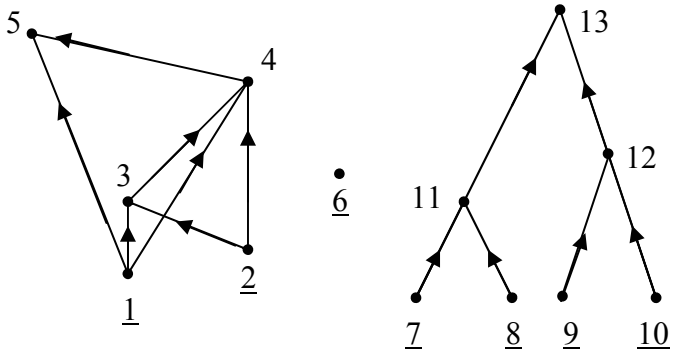
$A_{11}=\{7,8\}$; $A_{12}=\{9,10\}$; $A_{13}=\{11,12\}$.

Elementele din X au asociată valoarea 1.

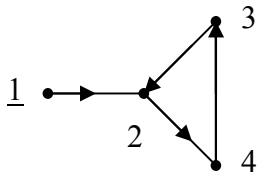
Fiecare funcție f_x calculează $v(x)$ ca fiind suma valorilor elementelor din A_x. Alegem $z=5$.

Este evident că vom obține $v=(1,1,2,4,5,1,1,1,1,1,2,2,4)$. O ordine posibilă de a considera elementele lui $A \setminus X$ astfel încât să putem calcula valoarea asociată lor este: $3,11,12,13,4,5$.

Lucrurile devin mai clare dacă reprezentăm problema pe un *graf de dependențe*. Vârfurile corespund elementelor din A, iar descendenții unui vârf x sunt vârfurile din A_x. Vârfurile din X apar subliniate.



Problema enunțată nu are totdeauna soluție, aşa cum se vede pe graful de dependențe de mai jos, în care există un circuit care nu permite calculul lui `v` în `z=3`.



Observații:
- `A` poate fi chiar infinită;
- `B` este de obicei **`N`**, **`Z`**, **`R`**, `{0,1}` sau un produs cartezian;
- f_x poate fi un minim, un maxim, o sumă etc.

Pentru orice `x∈A`, spunem că `x` este *accesibil* dacă, plecând de la `X`, poate fi calculată valoarea `v(x)`. Evident, problema are soluție dacă şi numai dacă `z` este accesibil.

Pentru orice `x∈A`, notăm prin O_x *mulțimea vârfurilor observabile din* `x`, adică mulțimea vârfurilor `y` pentru care există un drum de la `y` la `x`. Problema enunțată are soluție dacă şi numai dacă:
1) O_z nu are circuite;
2) vârfurile din O_z în care nu sosesc arce fac parte din `X`.

Prezentăm în continuare mai multe metode/încercări de rezolvare a problemei enunțate.

7.2. Metoda şirului crescător de mulţimi

Fie A o mulţime finită şi X o submulţime a sa. Definim următorul şir crescător de mulţimi:

$X_0 = X$

$X_{k+1} = X_k \cup \{x \in A \mid A_x \subset X_k\}, \forall k>0$

Evident, $X_0 \subset X_1 \subset \ldots \subset X_k \subset X_{k+1} \subset \ldots \subset A$.

Propoziţie. Dacă $X_{k+1}=X_k$, atunci $X_{k+i}=X_k$,$\forall i \in \mathbf{N}$.

Facem demonstraţia prin inducţie după i.

Pentru $i=1$ rezultatul este evident.

Presupunem $X_{k+i}=X_k$ şi demonstrăm că $X_{k+i+1}=X_k$:

X_{k+i+1} = cf. definiţiei şirului de mulţimi

$= X_{k+i} \cup \{x \in A \mid A_x \subset X_{k+i}\}$ = cf. ipotezei de inducţie

$= X_k \cup \{x \in A \mid A_x \subset X_k\}$ = cf. definiţiei şirului de mulţimi

$= X_{k+1}$ = cf. ipotezei

$= X_k$.

Consecinţe.

- ne oprim cu construcţia şirului crescător de mulţimi la primul k cu $X_k=X_{k+1}$ (A este finită!);
- dacă aplicăm cele de mai sus pentru problema generală enunţată, aceasta are soluţie dacă şi numai dacă $z \in X_k$.

Prezentăm în continuare algoritmul corespunzător acestei metode, adaptat la problema generală.

Vom lucra cu o partiţie $A=U \cup V$, unde U este mulţimea curentă de vârfuri a căror valoare asociată este cunoscută.

```
U ← X; V ← A\X
repeat
   W ←V
   for toţi x∈V
      if Ax⊂U
      then U ← U ∪{x}; V ←V\{x}
           calculează v(x) conform funcţiei fx
           if x=z
           then write v(x); stop
until V=W { nu s-a avansat! }
write(z, 'nu este accesibil')
```

Metoda descrisă are două deficienţe majore:
- la fiecare reluare se parcurg toate elementele lui V;
- nu este precizată o ordine de considerare a elementelor lui V.

Aceste deficienţe fac ca această metodă să nu fie performantă.

Metoda şirului crescător de mulţimi este larg folosită în teoria limbajelor formale, unde de cele mai multe ori ne interesează existenţa unui algoritm şi nu performanţele sale.

7.3. Sortarea topologică

Fie A={1,...,n} o mulţime finită. Pe A este dată o relaţie tranzitivă, notată prin "<". Relaţia este dată prin mulţimea perechilor (i,j) cu i<j.

Se cere să se listeze elementele 1,..,n ale mulţimii într-o ordine ce satisface cerinţa: dacă i<j, atunci i apare la ieşire înaintea lui j.

Problema enunţată apare, de exemplu, la înscrierea unor termeni într-un dicţionar astfel încât explicaţiile pentru orice termen să conţină numai termeni ce apar anterior.

Este evident că problema se transpune imediat la grafurile de dependenţă: se cere o parcurgere a vârfurilor grafului astfel încât dacă există un arc de la i la j, atunci i trebuie vizitat înaintea lui j.

Observaţii:
- problema are soluţie dacă şi numai dacă graful este aciclic;
- dacă există soluţie, ea nu este neapărat unică.

În esenţă, algoritmul care urmează repetă următorii paşi:
- determină i care nu are predecesori;
- îl scrie;
- elimină perechile pentru care sursa este i.

Fie M mulţimea curentă a vârfurilor care nu au predecesori. Iniţial M=X. Mulţimea M va fi reperezentată ca o coadă, notată cu C.

Pentru fiecare $i \in A$, considerăm:
S_i = lista succesorilor lui i;
$nrpred_i$ = numărul predecesorilor lui i din mulţimea M curentă.

Etapa de inițializare constă în următoarele:

```
Si ← Ø, nrpredi←0, ∀i
C ← Ø; nr ← 0    { nr este numărul elementelor produse la ieşire }
for k=1,m        { m este numărul perechilor din relația "<" }
   read(i,j)
   Si ⇐ j; nrpredj ← nrpredj+1
for i=1,n
   if nrpredi=0
   then i ⇒ C
```

Să observăm că timpul cerut de etapa de inițializare este de ordinul `O(m+n)`.

Algoritmul propriu-zis, adaptat la problema generală, este următorul:

```
while C ≠ Ø
   i ⇐ C; write(i); nr ← nr+1
   calculează v(i) conform funcției fi
   if i=z
   then write(i,v(i)); stop
   for toți j∈Si
      nrpredj ← nrpredj-1
      if nrpredj=0 then j ⇒ C
if nr<n then write('Nu')
```

Fiecare executare a corpului lui `while` necesită un timp proporțional cu $|S_i|$. Dar $|S_1|+...+|S_n|=m$, ceea ce face ca timpul de executare să fie de ordinul `O(m)`. Ținând cont şi de etapa de inițializare, rezultă că timpul total este de ordinul `O(m+n)`, deci liniar.

Totuşi, sortarea topologică aplicată problemei generale prezintă un dezavantaj: sunt calculate şi valori ale unor vârfuri "neinteresante", adică neobservabile din `z`.

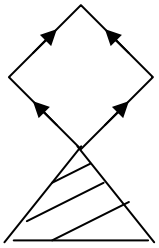
7.4. Încercare cu metoda Divide et Impera

Este folosită o procedură `DivImp`, apelată prin `DivImp(z)`.

```
procedure DivImp(x)
   for toți y∈Ax\X
      DivImp(y)
   calculează v(x) conform funcției fx
end;
```

Apare un avantaj: sunt parcurse doar vârfurile din O_z. Dezavantajele sunt însă decisive pentru renunțarea la această încercare:

- algoritmul nu se termină pentru grafuri ciclice;
- valoarea unui vârf poate fi calculată de mai multe ori, ca de exemplu pentru situația:



7.5. Soluție finală

Etapele sunt următoarele:

- identificăm G_z = subgraful asociat lui O_z ;
- aplicăm sortarea topologică.

Fie $G_z=(X_z,M_z)$. Inițial $X_z=\emptyset$, $M_z=\emptyset$.

Pentru a obține graful G_z executăm apelul `DF(z)`, unde procedura `DF` este:

```
procedure DF(x)
   x ⇒ Xz
   for toți y∈Ax
      if y∉Xz then (y,x) ⇒ Mz; DF(y)
end;
```

Timpul este liniar.

Observație. Ar fi totuşi mai bine dacă :

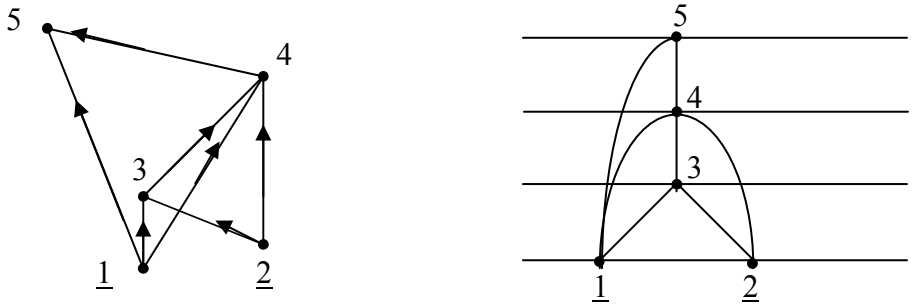
- am cunoaşte de la început G_z;
- forma grafului ar permite o parcurgere mai simplă.

7.6. Metoda programării dinamice

Definim un *PD–arbore de rădăcină* z ca fiind un graf de dependențe, aciclic, în care:

- $\forall x$, $x \in O_z$ (pentru orice vârf x există un drum de la x la z);
- $X=\{x \mid grad^-(x)=0\}$ (vârfurile în care nu sosesc arce sunt exact cele din submulțimea X).

Exemplu. Următorul graf este un PD-arbore de rădăcină `z=5`.



Un PD-arbore nu este neapărat un arbore, dar:

- poate fi pus pe niveluri: fiecare vârf `x` va fi pus pe nivelul egal cu lungimea celui mai lung
- drum de la `x` la `z`, iar sensul arcelor este de la nivelul inferior către cel superior;
- poate fi parcurs (cu mici modificări) în postordine;
  Prin parcurgerea în postordine, vârfurile apar sortate topologic.

Algoritmul de parcurgere în postordine foloseşte un vector `parcurs` pentru a ține evidența vârfurilor vizitate. Este inițializat vectorul `parcurs` şi se începe parcurgerea prin apelul `postord(z)`:

```
for toate vârfurile x∈A
   parcurs(x) ← x∈X
postord(z)
```

unde procedura `postord` cu argumentul `x` calculează `v(x)`:

```
procedure postord(x)
   for toți j∈Ax cu parcurs(j)=false
      postord(j)
   calculează v(x) conform funcției fx; parcurs(x)←true
end
```

Timpul de executare a algoritmului este evident liniar.

Metoda programării dinamice se aplică problemelor care urmăresc calcularea unei valori şi constă în următoarele:

1) Se asociază problemei un graf de dependențe;
2) În graf este pus în evidență un PD-arbore; problema se reduce la determinarea valorii asociate lui `z` (rădăcina arborelui);
3) Se parcurge în postordine PD-arborele.

Mai pe scurt, putem afirma că:

> *Metoda programării dinamice constă în identificarea unui PD-arbore şi parcurgerea sa în postordine.*

În multe probleme este util să căutăm în PD-arbore regularităţi care să evite memorarea valorilor tuturor vârfurilor şi/sau să simplifice parcurgerea în postordine.

Vom începe cu câteva exemple, la început foarte simple, dar care pun în evidenţă anumite caracteristici ale metodei programării dinamice.

Exemplul 1. Şirul lui Fibonacci

Ştim că acest şir este definit astfel:

$F_0=0;\ F_1=1;$

$F_n = F_{n-1} + F_{n-2}\ ,\ \forall n\geq 2$

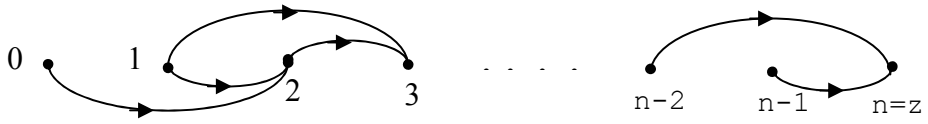
Dorim să calculăm F_n pentru un n oarecare.

Aici $A=\{0,...,n\}$, $X=\{0,1\}$, $B=\mathbf{N}$, iar

$A_k=\{k-1,k-2\},\ \forall k\geq 2$

$v(k)=F_k\ ;\ f_k(a,b)=a+b,\ \forall k\geq 2$

Un prim graf de dependenţe este următorul:

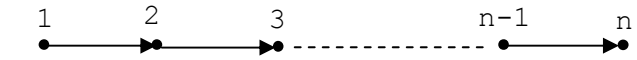


Să observăm că o mai bună alegere a mulţimii B simplifică structura PD-arborelui.

$A=\{1,2,...,n\};\ B=\mathbf{N}\times\mathbf{N};$

$v(k)=(F_{k-1},\ F_k);\quad f_k(a,b)=(b,a+b)$

$v(1)=(0,1).$



şi obţinem algoritmul binecunoscut:

```
a←0; b←1
for i=2,n
   (a,b)←(b,a+b)
write(b)
```

Exemplul 2. Calculul sumei $a_1+\ ...+a_n$

Este evident că trebuie calculate anumite sume parțiale.

O primă posibilitate este să considerăm un graf în care fiecare vârf să fie o submulțime $\{i_1,...,i_k\}$ a lui $\{1,2,...,n\}$, cu valoarea asociată $a_{i_1}+a_{i_2}+...+a_{i_k}$. Această abordare este nerealizabilă: numărul de vârfuri ar fi exponențial.

O a doua posibilitate este ca vârfurile să corespundă mulțimilor $\{i,i+1,...,j\}$ cu $i\le j$ şi cu valoarea ataşată $s_{i,j}=a_i+...+a_j$. Vom nota un astfel de vârf prin $(i:j)$. Dorim să calculăm valoarea $a_1+...+a_n$ vârfului $z=(1:n)$. Putem considera mai mulți PD-arbori:

- Arborele liniar constituit din vârfurile cu $i=1$. Obținem relațiile de recurență:

$s_{1,1}=a_1$;

$s_{1,j}=s_{1,j-1}+a_j$, $\forall j=2,3,...,n$

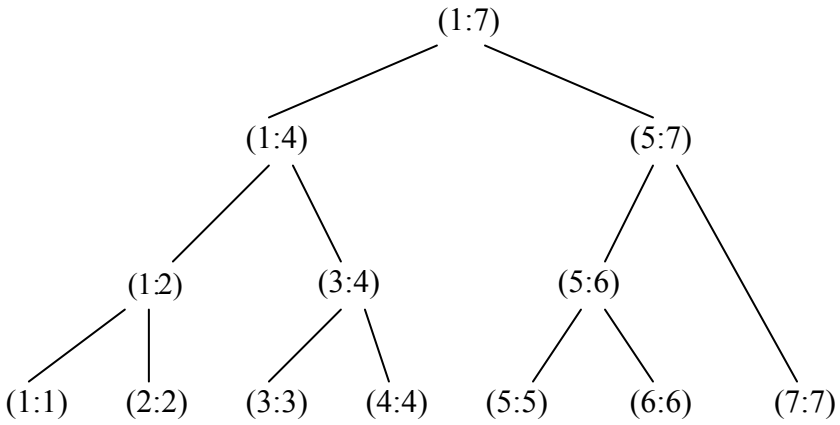
care corespund asociativității la stânga: $(...((a_1+a_2)+a_3)+...)$.

- Arborele liniar constituit din vârfurile cu $j=n$. Acest arbore corespunde asociativității la dreapta a sumei:

$s_{n,n}=a_n$

$s_{i,n}=a_i+s_{i-1,n}$, $\forall i=n-1,n-2,...1$.

- Arborele binar strict în care fiecare vârf (afară de frunze) are descendenții $(i:k)$ şi $(k+1:j)$ cu $k=\lfloor (i+j)/2 \rfloor$. Prezentăm acest arbore pentru $n=7$:



Relațiile de recurență sunt:

$s_{ii}=a_i$

$s_{i,j}=s_{ik}+s_{k+1,j}$ pentru $i<j$.

iar algoritmul constă în parcurgerea pe niveluri, de jos în sus, a arborelui; nu este folosit vreun tablou suplimentar:

```
k←1
while k<n
   k2←k+k; i←1
   while i+k≤n
      a_i←a_i+a_{i+k}; i←i+k2
   k ←k2
```

Rezultatul este obținut în a_1. Evoluția calculelor apare în următorul tabel:

n	k2	k	i	
7	2	1	1	$a_1 \leftarrow a_1+a_2$
			3	$a_3 \leftarrow a_3+a_4$
			5	$a_5 \leftarrow a_5+a_6$
			7	
	4	2	1	$a_1 \leftarrow a_1+a_3$
			5	$a_5 \leftarrow a_5+a_7$
			9	
	8	4	1	$a_1 \leftarrow a_1+a_5$
			9	

Algoritmul de mai sus nu este atât de stupid şi inutil pe cât apare la prima vedere pentru o problemă atât de simplă.

Într-adevăr, calculele pentru fiecare reluare a ciclului `while` interior sunt executate asupra unor seturi de date disjuncte. De aceea, în ipoteza că pe calculatorul nostru dispunem de mai multe procesoare, calculele pe fiecare nivel al arborelui (mergând de jos în sus) pot fi executate în paralel. Drept urmare, timpul de calcul va fi de ordinul $O(\log n)$, deci sensibil mai bun decât cel secvențial, al cărui ordin este $O(n)$.

Exemplul 3. Determinarea subşirului crescător de lungime maximă.

Se consideră vectorul $a=(a_1,\ldots,a_n)$. Se cer lungimea celui mai lung subşir crescător, precum şi toate subşirurile crescătoare de lungime maximă.

Introducem notațiile:

`nr` = lungimea maximă căutată;

`lung(i)` = lungimea maximă a subşirului crescător ce începe cu a_i.

$A=\{1,2,\ldots,n\}$; $X=\{n\}$;

$A_i=\{i+1,\ldots,n\}$ şi $f_i=lung(i)$, $\forall i<n$;

Evident, suntem în prezența unui PD-arbore de rădăcină 1.

Determinarea lui nr se face astfel:

```
nr ← 1; lung(n) ← 1
for i=n-1,1,-1
   lung(i) ← 1+max{lung(j)|j>i & ai<aj}
   nr ← max{nr,lung(i)}
```

Determinarea tuturor subşirurilor crescătoare de lungime maximă se face printr-un backtracking recursiv optimal. Subşirurile se obțin în vectorul s, iar ind reprezintă ultima poziție completată din s.

```
for i=1,n
   if lung(i)=nr
   then ind ←1; s(1)←ai; scrie(i)
```

unde procedura scrie are forma:

```
procedure scrie(i)
   if ind=nr
   then write(s)
   else for j=i+1,n
          if ai<aj & lung(i)=1+lung(j)
          then ind←ind+1; s(ind)←a(j); scrie(j); ind←ind-1
end;
```

Exemplul 4. Înmulțirea optimă a unui şir de matrici.

Avem de calculat produsul de matrici $A_1 \times A_2 \times ... \times A_n$, unde dimensiunile matricilor sunt respectiv $(d_1,d_2),(d_2,d_3),....,(d_n,d_{n+1})$. Ştiind că înmulțirea matricilor este asociativă, se pune problema ordinii în care trebuie înmulțite matricile astfel încât numărul de înmulțiri elementare să fie minim.

Presupunem că înmulțirea a două matrici se face în modul uzual, adică produsul matricilor A(m,n) şi B(n,p) necesită $m \times n \times p$ înmulțiri elementare.

Pentru a pune în evidență importanța ordinii de înmulțire, să considerăm produsul de matrici $A_1 \times A_2 \times A_3 \times A_4$ unde $A_1(100,1)$, $A_2(1,100)$, $A_3(100,1)$, $A_4(1,100)$.

Pentru ordinea de înmulțire $(A_1 \times A_2) \times (A_3 \times A_4)$ sunt necesare 1.020.000 de înmulțiri elementare. În schimb, pentru ordinea de înmulțire $(A_1 \times (A_2 \times A_3)) \times A_4$ sunt necesare doar 10.200 de înmulțiri elementare.

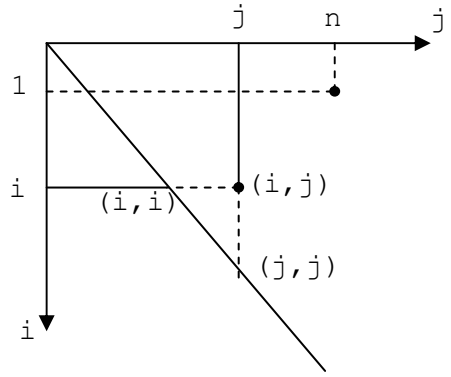
Fie `cost(i,j)` numărul minim de înmulţiri elementare pentru calculul produsului $A_i \times ... \times A_j$. Punând în evidenţă ultima înmulţire de matrici, obţinem relaţiile:

$cost(i,i) = 0, \ \forall i=1,2,...,n$

$cost(i,j) = \min \{cost(i,k)+cost(k+1,j)+d_i \times d_{k+1} \times d_{j+1} \mid i \le k < j\}$.

Valoarea cerută este `cost(1,n)`.

Vârfurile grafului de dependenţă sunt perechile `(i,j)` cu $i \le j$. Valoarea `cost(i,j)` depinde de valorile vârfurilor din stânga şi de cele ale vârfurilor de deasupra. Se observă uşor că suntem în prezenţa unui PD-arbore.



Forma particulară a PD-arborelui nu face necesară aplicarea algoritmului general de parcurgere în postordine: este suficient să parcurgem în ordine coloanele 2,...,n, iar pe fiecare coloană `j` să mergem în sus de la diagonală până la `(i,j)`.

```
for j=2,n
   for i=j-1,1,-1
      cost(i,j) calculat ca mai sus; fie k valoarea pentru care se realizează
        minimul
      cost(j,i)←k
write cost(1,n)
```

(se observă că am folosit partea inferior triunghiulară a matricii pentru a memora indicii pentru care se realizează minimul).

Dacă dorim să producem şi o ordine de înmulţire optimă, vom apela `sol(1,n)`, unde procedura `sol` are forma:

```
procedure sol(p,u)
   if p=u
   then write(p)
```

```
   else k←cost(u,p)
        write('('); sol(p,k); write(',');
        sol(k+1,u); write(')')
end;
```

Pentru evaluarea timpului de lucru, vom calcula numărul de comparări efectuate. Aceste este:

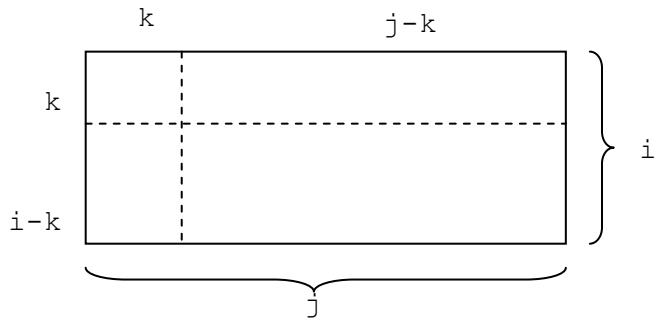
$$\sum_{j=2}^{n}\sum_{i=1}^{j-1}(j-i+1)=\sum_{j=2}^{n}\left[j(j-1)-\frac{(j-1)(j-2)}{2}\right]=O(n^3)$$

Exemplul 5. Descompunerea unui dreptunghi în pătrate

Se consideră un dreptunghi cu laturile de m, respectiv n unități ($m<n$). Asupra sa se pot face tăieturi *complete* pe orizontală sau verticală. Se cere numărul minim de pătrate în care poate fi descompus dreptunghiul.

Fie a_{ij} = numărul minim de pătrate în care poate fi descompus un dreptunghi de laturi i şi j. Evident $a_{ij}=a_{ji}$. Rezultatul căutat este a_{mn}.

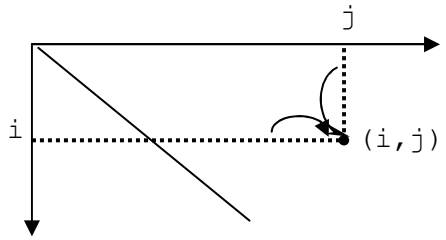
Vârfurile grafului de dependențe sunt (i,j), iar valorile asociate sunt a_{ij}.



Pentru calculul lui a_{ij} avem de ales între a face:

- o tăietură pe verticală; costurile sunt: $a_{ik}+a_{i,j-k}$, $k\le \lfloor j/2 \rfloor$;
- o tăietură pe orizontală; costurile sunt: $a_{k,j}+a_{i-k,j}$, $k\le \lfloor i/2 \rfloor$.

Rezultă că valoarea a_{ij} a unui vârf (i,j) depinde de valorile vârfurilor din stânga sa şi de cele aflate deasupra sa. Se observă că graful de dependențe este un PD-arbore.



Dependențele pot fi exprimate astfel:

$a_{i,1}=i, \ \forall i=1,...,m$

$a_{1,j}=j, \ \forall j=1,...,n$

$a_{ii}=1, \ \forall=1,...,m$

$a_{ij} = \min\{\alpha,\beta\}$, unde

$\alpha=\min\{a_{ik}+a_{i,j-k} \mid k\leq \lfloor j/2 \rfloor\}$, iar $\beta=\min\{a_{k,j}+a_{i-k,j} \mid k\leq \lfloor i/2 \rfloor\}$.

Forma particulară a PD-arborelui permite o parcurgere mai uşoară decât aplicarea algoritmului general de postordine. De exemplu putem coborî pe linii, iar pe fiecare linie mergem de la stânga la dreapta.

După inițializările date de primele trei dependențe de mai sus, efectuăm calculele:

```
for i=2,m
   for j=i+1,n
      calculul lui a_ij conform celei de a patra dependențe de mai sus
      if j≤m then a_ji←a_ij
```

Observație. Am lucrat numai pe partea superior triunghiulară, cu actualizări dedesubt.

8 METODA BRANCH AND BOUND

8.1. Prezentare generală

Metoda *Branch and Bound* se aplică problemelor care care pot fi reprezentate pe un arbore: se începe prin a lua una dintre mai multe decizii posibile, după care suntem puşi în situaţia de a alege din nou dintre mai multe decizii; vom alege una dintre ele etc. Vârfurile arborelui corespund stărilor posibile în dezvoltarea soluţiei.

Deosebim două tipuri de probleme:

1) Se caută un anumit vârf, numit *vârf rezultat*, care evident este final (nu are descendenţi).
2) Există mai multe vârfuri finale, care reprezintă soluţii posibile, dintre care căutăm de exemplu pe cel care *minimizează* o anumită funcţie.

Exemplul 1. Jocul 15 (Perspico).

Un număr de 15 plăcuţe pătrate sunt incorporate într-un cadru 4×4, o poziţie fiind liberă. Fiecare plăcuţă este etichetată cu unul dintre numerele 1,2,...,15. Prin *configuraţie* înţelegem o plasare oarecare a plăcuţelor în cadru. Orice plăcuţă adiacentă cu locul liber poate fi mutată pe acest loc liber. Dându-se o configuraţie iniţială şi una finală, se cere să determinăm o succesiune de mutări prin care să ajungem din configuraţia iniţială în cea finală. Configuraţiile iniţială şi finală pot fi de exemplu:

1	2	3	4
5		7	8
9	6	10	11
13	14	15	12

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

unde locul liber mai poate fi considerat drept conţinând plăcuţa imaginară cu eticheta 16.

Observaţie. Mutarea unei plăcuţe adiacente locului liber pe acel loc poate fi gândită şi ca mutarea locului liber pe o poziţie adiacentă.

Prezentăm întâi o condiție de existență a unei succesiuni de mutări prin care se poate trece de la configurația inițială în cea finală.

Cele 16 locaşuri sunt considerate ca fiind ordonate de la stânga la dreapta şi de jos în sus. Pentru plăcuța etichetată cu `i` definim valoarea `n(i)` ca fiind numărul locaşurilor care urmează celei pe care se află plăcuța şi care conțin o plăcuță a cărei etichetă este mai mică decât `i`. De exemplu pentru configurația inițială de mai sus avem:

`n(8)=1; n(4)=0; n(16)=10; n(15)=1` etc.

Fie `l` şi `c` linia şi coloana pe care apare locul liber. Fie `x∈{0,1}` definit astfel: `x=0` dacă şi numai dacă `l+c` este par. Se poate demonstra următorul rezultat:

Propoziție. Fiind dată o configurație inițială, putem trece din ea la configurația finală de mai sus ⇔ `n(1)+n(2)+...+n(16)+x` este par.

În continuare vom presupune că putem trece de la configurația inițială la cea finală.

Cum locul liber poate fi mutat spre N, S, E, V (fără a ieşi însă din cadru), rezultă că fiecare configurație (stare) are cel mult 4 descendenți. Se observă că arborele astfel construit este infinit. Stările finale sunt stări rezultat şi corespund configurației finale.

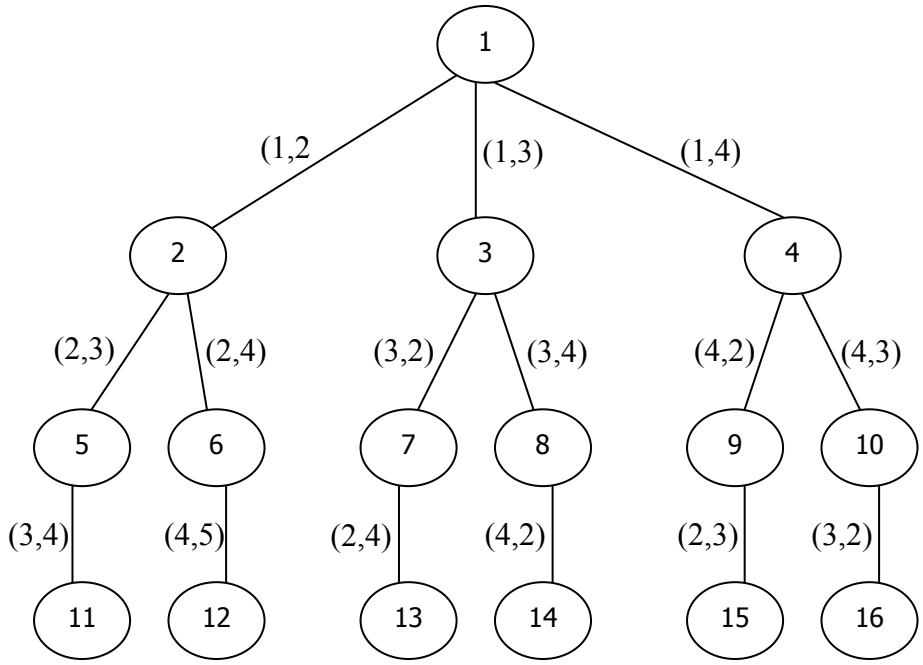
Exemplul 2. Circuitul hamiltonian de cost minim.

Se consideră un graf orientat cu arcele etichetate cu costuri pozitive. Inexistența unui arc între două vârfuri este identificată prin "prezența" sa cu costul $+\infty$. Presupunem că graful este dat prin matricea `C` a costurilor sale. Se cere să se determine, dacă există, un circuit hamiltonian de cost minim.

Să considerăm, de exemplu, graful dat de matricea de costuri:

$$C = \begin{pmatrix} \infty & 3 & 7 & 2 \\ 5 & \infty & 1 & 9 \\ 4 & 8 & \infty & 3 \\ 6 & 2 & 6 & \infty \end{pmatrix}$$

Arborele spațiului de stări, în care muchiile corespund arcelor din graf, este următorul:



subînțelegându-se că se pleacă din vârful 1 şi că din frunze se revine la acest vârf.

Revenim la descrierea metodei Branch and Bound.

Ştim că şi metoda backtracking este aplicabilă problemelor reprezentabile pe arbori. Există însă multe deosebiri, dintre care menționăm următoarele:

- ordinea de parcurgere a arborelui;
- modul în care sunt eliminați subarborii care nu pot conduce la o soluție;
- faptul că arborele poate fi infinit (prin natura sa sau prin faptul că mai multe vârfuri pot corespunde la o aceeaşi stare).

În general arborele de stări este construit dinamic.

Este folosită o listă `L` de *vârfuri active*, adică de stări care sunt susceptibile de a fi dezvoltate pentru a ajunge la soluție/soluții. Inițial, lista `L` conține rădăcina arborelui, care este *vârful curent*. La fiecare pas, din `L` alegem un vârf (care nu este neapărat un fiu al vârfului curent!), care devine noul vârf curent.

Când un vârf activ devine vârf curent, sunt generați toți fiii săi, care devin vârfuri active (sunt incluşi în `L`). Apoi din nou este selectat un vârf curent.

Legat de modul prin care alegem un vârf activ drept vârf curent, deci implicit legat de modul de parcurgere a arborelui, facem următoarele remarci:

– parcurgerea DF nu este adecvată, deoarece pe de o parte arborele poate fi infinit, iar pe de altă parte soluția căutată poate fi de exemplu un fiu al rădăcinii

diferit de primul fiu şi parcurgerea în adâncime ar fi ineficientă: se parcurg inutil stări, în loc de a avansa direct spre soluție;
– parcurgerea pe lățime conduce totdeauna la soluție (dacă aceasta există), dar poate fi ineficientă dacă vârfurile au mulți fii.

Metoda Branch and Bound încearcă un "compromis" între cele două parcurgeri menționate mai sus, ataşând vârfurilor active câte un *cost* pozitiv, ce intenționează să fie o măsură a gradului de "apropiere" a vârfului de o soluție. Alegerea acestui cost este decisivă pentru a obține un timp de executare cât mai bun şi depinde de problema concretă, dar şi de abilitatea programatorului.

Observație. Costul unui vârf va fi totdeauna mai mic decât cel al descendenților (fiilor) săi.

De fiecare dată drept vârf curent este ales cel de cost minim (cel considerat ca fiind cel mai "aproape" de soluție). De aceea `L` va fi în general un min-ansamblu: costul fiecărui vârf este mai mic decât costul descendenților.

Din analiza teoretică a problemei deducem o valoare `lim` care este o aproximație prin adaos a minimului căutat: atunci când costul unui vârf depăşeşte `lim`, vârful curent este ignorat: nu este luat în considerare şi deci este eliminat întregul subarbore pentru care este rădăcină. Dacă nu cunoaştem o astfel de valoare `lim`, o inițializăm cu $+\infty$.

Se poate defini o *funcție de cost ideală*, pentru care `c(x)` este dat de:
- nivelul pe care se află vârful `x` dacă `x` este vârf rezultat;
- $+\infty$ dacă `x` este vârf final, diferit de vârf rezultat;
- min {`c(y)` | `y` fiu al lui `x`} dacă `x` nu este vârf final.

Această funcție este ideală din două puncte de vedere:
- nu poate fi calculată dacă arborele este infinit; în plus, chiar dacă arborele este finit, el trebuie parcurs în întregime, ceea ce este exact ce dorim să evităm;
- dacă totuşi am cunoaşte această funcție, soluția poate fi determinată imediat: plecăm din rădăcină şi coborâm mereu spre un vârf cu acelaşi cost, până ajungem în vârful rezultat.

Neputând lucra cu funcția ideală de mai sus, vom alege o aproximație $\hat{c}$ a lui `c`, care trebuie să satisfacă condițiile:
1) în continuare, dacă `y` este fiu al lui `x` avem $\hat{c}$`(x)`$<$$\hat{c}$`(y)`;
2) $\hat{c}$`(x)` să poată fi calculată doar pe baza informaților din drumul de la rădăcină la `x`;

3) este indicat ca $\hat{c} \le c$ pentru a ne asigura că dacă $\hat{c}(x) > lim$, atunci şi $c(x) > lim$, deci x nu va mai fi dezvoltat.

O primă modalitate de a asigura compromisul între parcurgerile în adâncime şi pe lățime este de a alege funcția $\hat{c}$ astfel încât, pentru o valoare naturală k, să fie îndeplinită condiția: pentru orice vârf x situat pe un nivel nx şi orice vârf situat pe un nivel $ny \ge nx+k$, să avem $\hat{c}(x) > \hat{c}(y)$, indiferent dacă y este sau nu descendent al lui x.

Condiția de mai sus spune că niciodată nu poate deveni activ un vârf aflat pe un nivel $ny \ge nx+k$ dacă în L apare un vârf situat pe nivelul nx, adică nu putem merge "prea mult" în adâncime. Dacă această condiție este îndeplinită, este valabilă următoarea propoziție:

Propoziție. În ipoteza că este îndeplinită condiția de mai sus şi dacă există soluție, ea va fi atinsă într-un timp finit, chiar dacă arborele este infinit.

Putem aplica cele de mai sus pentru jocul Perspico, alegând:
$\hat{c}(x)$ = suma dintre lungimea drumului de la rădăcină la x şi numărul de plăcuțe care nu sunt la locul lor (aici k=15).

8.2. Algoritmul Branch & Bound pentru probleme de optim

Să presupunem că dorim să determinăm vârful final de cost minim şi drumul de la rădăcină la el. Fie `lim` aproximarea prin adaos considerată mai sus.

Algoritmul este următorul (`rad` este rădăcina arborelui, iar i_{final} este vârful rezultat):

```
i ← rad; L ⇐ {i}; min ← lim;
calculăm ĉ(rad); tata(i) ← 0
while L ≠ ∅
    i ⇐ L {este scos vârful i cu ĉ(i) minim din min-ansamblul L}
    for toți j fii ai lui i
        calculăm ĉ(j); calcule locale asupra lui j; tata(j) ← i
        if j este vârf final
        then if ĉ(j)<min
             then min ← ĉ(j); i_final ← j
                  elimină din L vârfurile k cu ĉ(k)≥min   (*)
        else if ĉ(j)<min
             then j ⇒ L
```

```
if min=lim then write('Nu există soluție')
           else writeln(min); i ← i_final
                while i ≠ 0
                   write(i); i ← tata(i)
```

Observație. La `(*)` am ținut cont de faptul că dacă `j` este descendent al lui `i`, atunci $\hat{c}(i) < \hat{c}(j)$.

Vom aplica algoritmul de mai sus pentru problema circuitului hamiltonian de cost minim, pe exemplul considerat mai sus.

Pentru orice vârf `x` din arborele de stări, valoarea `c(x)` dată de funcția de cost ideală este:

- lungimea circuitului corespunzător lui `x` dacă `x` este frunză
- min $\{c(y) \mid y$ fiu al lui $x\}$ altfel.

Fiecărui vârf `x` îi vom ataşa o *matrice de costuri* M_x (numai dacă nu este frunză) şi o valoare $\hat{c}(x)$.

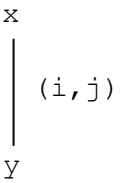
Observație. Dacă micşorăm toate elementele unei linii sau coloane cu α, orice circuit hamiltonian va avea costul micşorat cu α, deoarece în orice circuit hamiltonian din orice vârf pleacă exact un arc şi în orice vârf soseşte exact un arc.

Conform acestei observații, vom lucra cu *matrici de costuri reduse* (în care pe orice linie sau coloană apare cel puțin un zero, exceptând cazul când linia sau coloana conține numai ∞).

Pentru rădăcina `rad=1` plecăm de la matricea de costuri `C`. Matricea ataşată va fi matricea redusă obținută din `C`, iar $\hat{c}(1)$ = cantitatea cu care s-a redus matricea `C`.

În general, pentru un vârf `y` oarecare al cărui tată este `x` şi muchia `(x,y)` este etichetată cu `(i,j)`:

- dacă `y` este vârf terminal, $\hat{c}(x)$ va fi chiar `c(y)`, adică costul real al circuitului;
- în caz contrar, plecând de la M_x şi $\hat{c}(x)$ procedăm astfel:
  - elementele liniei `i` devin ∞, deoarece mergem sigur către vârful `j` din graf;
  - elementele coloanei `j` devin ∞, deoarece am ajuns sigur în vârful `j` din graf;
  - $M_x(j,1) \leftarrow \infty$, pentru a nu reveni prematur în rădăcina 1;
  - reducem noua matrice M_x şi obținem M_y; fie `r` cantitatea cu care s-a redus M_x. Vom lua $\hat{c}(y) \leftarrow \hat{c}(x)+r+M_x(i,j)$.



Concret, pentru exemplul dat, calculele se desfăşoară astfel:

- Pentru rădăcină:
  - reducem liniile în ordine cu 2, 1, 3, 2;
  - reducem prima coloană cu 1;
  - în acest mod obţinem $\hat{c}(1)=9$

$$M_1=\begin{pmatrix}\infty & 1 & 5 & 0\\ 3 & \infty & 0 & 8\\ 0 & 5 & \infty & 0\\ 3 & 0 & 4 & \infty\end{pmatrix}$$

♦ Acum $\min\leftarrow 9$; $L=\{1\}$. Este extras vârful 1 şi sunt consideraţi fiii săi.

- Pentru vârful 2:
  - plecăm de la M_1 şi punem ∞ pe linia 1 şi coloana 2;
  - elementul de pe linia 2 şi coloana 1 devine ∞;
  - reducem linia 3 cu 3;
  - în acest mod obţinem $\hat{c}(2)=9+3+1=13$

$$M_2=\begin{pmatrix}\infty & \infty & \infty & \infty\\ \infty & \infty & 0 & 8\\ 0 & \infty & \infty & 0\\ 0 & \infty & 1 & \infty\end{pmatrix}$$

- Pentru vârful 3:
  - plecăm de la M_1 şi punem ∞ pe linia 1 şi coloana 3;
  - elementul de pe linia 3 şi coloana 1 devine ∞;
  - reducem linia 2 cu 3;
  - în acest mod obţinem $\hat{c}(3)=9+3+5=17$

$$M_3=\begin{pmatrix}\infty & \infty & \infty & \infty\\ 0 & \infty & \infty & 5\\ \infty & 5 & \infty & 0\\ 3 & 0 & \infty & \infty\end{pmatrix}$$

- Pentru vârful 4:
  - plecăm de la M_1 şi punem ∞ pe linia 1 şi coloana 4;
  - elementul de pe linia 4 şi coloana 1 devine ∞;
  - nu este necesară vreo reducere;
  - în acest mod obţinem $\hat{c}(4)=9+0+0=9$

$$M_4=\begin{pmatrix}\infty & \infty & \infty & \infty\\ 3 & \infty & 0 & \infty\\ 0 & 5 & \infty & \infty\\ \infty & 0 & 4 & \infty\end{pmatrix}$$

♦ Acum $L=\{2,3,4\}$ cu $\hat{c}(2)=13$, $\hat{c}(3)=17$, $\hat{c}(4)=9$. Devine activ vârful 4.

- Pentru vârful 9:
  - plecăm de la M_4 şi punem ∞ pe linia 4 şi coloana 2;
  - elementul de pe linia 2 şi coloana 1 devine ∞;
  - nu este necesară vreo reducere;
  - în acest mod obţinem $\hat{c}(9)=9+0+0=9$

$$M_9=\begin{pmatrix}\infty & \infty & \infty & \infty\\ \infty & \infty & 0 & \infty\\ 0 & \infty & \infty & \infty\\ \infty & \infty & \infty & \infty\end{pmatrix}$$

- Pentru vârful 10:
  - plecăm de la M_4 şi punem ∞ pe linia 4 şi coloana 3;
  - elementul de pe linia 3 şi coloana 1 devine ∞;
  - reducem linia 2 cu 3, iar linia 3 cu 5;
  - în acest mod obţinem $\hat{c}(10)=9+8+4=21$

$$M_{10}=\begin{pmatrix}\infty & \infty & \infty & \infty\\ 0 & \infty & \infty & \infty\\ \infty & 0 & \infty & \infty\\ \infty & \infty & \infty & \infty\end{pmatrix}$$

- Acum $L=\{2,3,9,10\}$ cu $\hat{c}(2)=13$, $\hat{c}(3)=17$, $\hat{c}(9)=9$, $\hat{c}(10)=21$. Devine activ vârful 9. Singurul său descendent este 15, care este frunză. $\hat{c}(15)=c(15)=9$ (costul real al circuitului). Sunt eliminate din L vârfurile cu costurile mai mari decât 9, deci L devine vidă. $\min$ rămâne egal cu 9, va fi produs la ieşire circuitul căutat (1,4,2,3,1) şi algoritmul se opreşte.

9 DRUMURI ÎN GRAFURI

Fie $G=(V,M)$ graf orientat cu $n=|V|$, $m=|M|$. Fie A matricea sa de adiacență.

Considerăm şirul de matrici:

$$\begin{cases} A^1 = A \\ A^k = A^{k-1} \cdot A, \ \forall k \geq 2 \end{cases}$$

a cărui semnificație este următoarea:

Propoziția 1. $A^k(i,j)$ = numărul drumurilor de lungime k de la i la j.

- pentru $k=1$: evident.
- $k-1 \rightarrow k$: $A^k(i,j) = \sum_{s=1}^{n} A^{k-1}(i,s) \cdot A(s,j)$, unde s este penultimul vârf din drumul de la i la j; pentru fiecare s cu $A(s,j)=1$, la sumă se adaugă numărul drumurilor de lungime $k-1$ de la i la s, adică numărul drumurilor de lungime k de la i la j având pe s ca penultim vârf.

În continuare dorim să determinăm numai *existența* drumurilor de lungime k. Considerăm şirul de matrici:

$$\begin{cases} A^{(1)} = A \\ A^{(k)} = A^{(k-1)} \circ A, \ \forall k \geq 2 \end{cases}$$

unde $A^{(k)}(i,j) = \bigvee_{s=1}^{n} A^{(k-1)}(i,s) \wedge A^{(k-1)}(s,j)$

a cărui semnificație este următoarea (elementele matricilor sunt 0 sau 1):

Propoziția 2. $A^{(k)}(i,j)=1 \Leftrightarrow$ există drum de lungime k de la i la j.

Demonstrația se face prin inducție ca mai sus.

Definim *matricea drumurilor* D prin:

$D(i,j)=1 \Leftrightarrow \exists$ drum de la i la j.

$D=A^{(1)} \vee \ldots \vee A^{(n-1)}$, deoarece dacă există un drum de la i la j, există şi un drum de lungime cel mult egală cu $n-1$ de la i la j.

Construcțiile matricilor de mai sus necesită un timp de ordinul $O(n^4)$.

Vom căuta să obținem un timp de executare mai bun, inclusiv pentru cazul în care lungimea arcelor este oarecare (în cele de mai sus s-a presupus implicit că arcele au lungimea egală cu 1).

În continuare, fiecare arc `<i,j>` va avea o etichetă `et(<i,j>)` strict pozitivă, ce reprezintă lungimea arcului.

Considerăm $P(i,j)=\begin{cases} et(<i,j>) & \text{dacă } <i,j>\in M \\ 0 & \text{dacă } i=j \\ +\infty & \text{altfel} \end{cases}$

şi şirul de matrici:

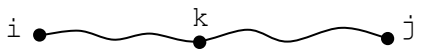
$$\begin{cases} P_0 = P \\ P_k(i,j) = \min\{P_{k-1}(i,j), P_{k-1}(i,k)+P_{k-1}(k,j)\}, \quad \forall k \geq 1 \end{cases}$$

Propoziția 3. P_n este matricea celor mai scurte drumuri.

Vom demonstra prin inducție după `k` următoarea afirmație:

$P_k(i,j)$ = lungimea celui mai scurt drum de la `i` la `j` în care numerele de ordine ale nodurilor *intermediare* sunt cel mult egale cu `k`.

- pentru `k=0`: evident (nu există vârfuri intermediare).
- `k-1` $\rightarrow$ `k` : Considerăm un drum de lungime minimă de la `i` la `j`.
  Dacă drumul nu trece prin `k`, $P_k(i,j)=P_{k-1}(i,j)$.
  Dacă drumul trece prin `k`, el va trece o singură dată prin `k` (are lungime minimă) şi în drumurile de lungime minimă de la `i` la `k` şi de la `k` la `j` vârful `k` nu apare ca vârf intermediar, deci $P_k(i,j)=P_{k-1}(i,k)+P_{k-1}(k,j)$.



Observații:

1) s-a folosit metoda programării dinamice;
2) $P_k(i,i)=0$;
3) $P_k(i,k)=P_{k-1}(i,k)$ şi $P_k(k,j)=P_{k-1}(k,j)$, deci la trecerea de la P_{k-1} la P_k linia `k` şi coloana `k` rămân neschimbate.

Rezultă că putem folosi o singură matrice. Ajungem astfel la *algoritmul Floyd-Warshall*:

```
for k=1,n
   for i=i,n
      for j=1,n
          P(i,j) ← min {P(i,j),P(i,k)+P(k,j)}
```

Timpul de executare este evident de ordinul $O(n^3)$.

Dacă dorim să determinăm doar existența drumurilor şi nu lungimea lor minimă, vom proceda similar. Considerăm şirul de matrici:

$$\begin{cases} A_0 = A \\ A_k(i,j) = A_{k-1}(i,j) \vee [A_{k-1}(i,k) \wedge A_{k-1}(k,j)] \quad , \forall k \geq 0 \end{cases}$$

Propoziția 4. A_n este matricea drumurilor.

Demonstrăm prin inducție după k următoarea afirmație:
$A_k(i,j)=1 \Leftrightarrow \exists$ drum de la i la j cu numerele de ordine ale vârfurilor intemediare egale cu cel mult k.

- pentru $k=0$: evident;
- $k-1 \rightarrow k$:
  Dacă $A_k(i,j)=1$, atunci fie $A_{k-1}(i,j)=1$, fie $A_{k-1}(i,k)=A_{k-1}(k,j)=1$; în ambele situații va exista, conform ipotezei de inducție, un drum de la i la j cu numerele de ordine ale vârfurilor intemediare egale cu cel mult k.
  Dacă există un drum de la i la j cu numerele de ordine ale vârfurilor intemediare egale cu cel mult k, prin eliminarea ciclurilor drumul va trece cel mult o dată prin k. Este suficient în continuare să considerăm cazul în care drumul trece prin vârful k şi cazul în care drumul nu trece prin k.

Sunt valabile aceleaşi observații ca la *Propoziția 3*, iar algoritmul are o formă similară:

```
D←A
for k=1,n
   for i=1,n
      for j=1,n
         D(i,j) ← D(i,j) ∨ [D(i,k)∧D(k,j)]
```

Timpul de executare este evident de ordinul $O(n^3)$.

În continuare ne vor interesa numai drumurile ce pleacă dintr-un vârf x_0 fixat. Este de aşteptat ca timpul de executare să scadă.

Mai precis, căutăm $d(x)$ = lungimea drumului minim de la x_0 la x, pentru orice vârf x. În plus, dorim să determinăm şi câte un astfel de drum.

Prezentăm în continuare *algoritmul lui Dijkstra* pentru problema enunțată. Pentru simplificare, presupunem că orice vârf este accesibil din x_0.

Pentru regăsirea drumurilor vom folosi vectorul $tata$.

Perechiile $(d(x),tata(x))$ sunt inițializate astfel:

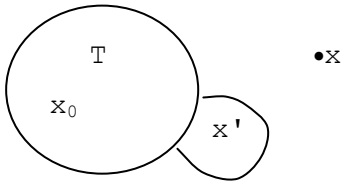
- $(0,0)$ pentru $x=x_0$;
- $(et(<x_0,x>),x_0)$ dacă $<x_0,x> \in M$;
- $(+\infty,0)$ altfel.

Fie T = mulțimea vârfurilor x pentru care $d(x)$ are valoarea finală.
În continuare, algoritmul lucrează astfel:

```
T ← {x0}
while T≠V
   Fie x'∈V\T cu d(x') minim
   T ← T∪{x'}
   for toți x∉T
       if d(x)>d(x')+et(<x',x>)
       then d(x) ← d(x')+et(<x',x>); tata(x)←x'     (*)
```

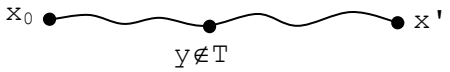
Pentru a demonstra corectitudinea algoritmului, vom arăta prin inducție după $|T|$ că:

1) $\forall\ x\in T : d(x)$ = lungimea celui mai scurt drum de la x_0 la x;
2) $\forall\ x\notin T : d(x)$ = lungimea celui mai scurt drum de la x_0 la x, ce trece numai prin vârfuri din T.



Pentru $|T|=1$, concluzia este evidentă.
$|T| \rightarrow |T|+1$: Fie x' vârful nou adăugat.
Demonstrăm cele două afirmații de mai sus:

1) $d(x')$ este cel final:
Presupunem prin absurd că există un drum de la x_0 la x' de lungime mai mică decât $d(x')$. Acest drum trebuie să treacă printr-un vârf $y\notin T$.



Evident $d(y)<d(x')$. Contradicție, pentru că a fost ales x' cu $d(x')$ minim.

2) Evident, conform actualizărilor $(*)$ efectuate de algoritm.

Observații.

1) Timpul de executare este de ordinul $O(n^2)$.
2) Pe baza vectorului $tata$, putem regăsi pentru orice $x\in V$ drumul minim ce îl leagă de x_0 în timp liniar. Regăsirea tuturor acestor drumuri necesită un timp de ordinul $O(n^2)$, deci complexitatea în timp a algoritmului nu creşte.
3) Dacă dorim să aflăm numai drumul minim de la x_0 la un vârf x_1 dat, ne oprim când $x'=x_1$; aceasta nu implică însă o reducere a ordinului de mărime al timpului de calcul.
4) Algoritmul de mai sus poate fi încadrat la metoda Greedy, dar şi la metoda şirului crescător de mulțimi.