

5. Recursivitate

5.1. Introducere

5.1.1. Definiție. Iterație și recursivitate

- O definiție **recursivă** este acea **definiție** care se referă la **un obiect** care **se definește ca parte a propriei sale definiri**.
 - Desigur o definiție de genul "*o floare este o floare*" care poate reprezenta în poezie un univers întreg, în știință în general și în matematică în special **nu** furnizează prea multe informații despre floare.
- O caracteristică foarte importantă a **recursivității** este aceea de a preciza o definiție într-un **sens evolutiv**, care evită circularitatea.
 - Spre exemplu o **definiție recursivă** este următoarea: "*Un buchet de flori este: (1) fie o floare, (2) fie o floare adăugată buchetului*".
 - Afirmatia (1) servește ca și **condiție inițială**, indicând maniera de amorsare a definiției.
 - Afirmatia (2) precizează **definirea recursivă (evolutivă) propriu-zisă**.
 - **Varianta iterativă** a aceleiași definiții este "*Un buchet de flori constă fie dintr-o floare, fie din două, fie din 3 flori, fie ... etc*".
 - După cum se observă, definiția recursivă este **simplică și elegantă** dar oarecum indirectă, în schimb definiția iterativă este directă dar greoaie și lipsită de eleganță.
- Despre un **obiect** se spune că este **recursiv** dacă el **constă** sau **este definit prin el însuși**.
 - Prin **definiție** orice **obiect recursiv** implică **recursivitatea** ca și proprietate **intrinsecă** a obiectului în cauză.
- **Recursivitatea** este utilizată cu multă eficiență în **matematică**, spre exemplu în definirea numerelor naturale, a structurilor arbore sau a anumitor funcții [5.1.1.a].

-
- **Numerele naturale:**

(1) 1 este un număr natural.

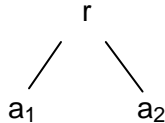
(2) Succesorul unui număr natural este un număr natural.

- **Structura arbore binar:**

(1) Φ este un arbore binar fără nici un nod, numit arbore binar vid.

(2) r este un arbore binar cu un singur nod.

(3) Dacă a_1 și a_2 sunt arbori binari, atunci structura



este un arbore binar (reprezentat răsturnat).

[5 . 1 . 1 . a]

- **Funcția factorial** $n!$ (definită pentru întregi pozitivi):

(1) $0! = 1$

(2) Dacă $n > 0$, atunci $n! = n * (n-1) !$

- Puterea **recursivității** rezidă evident în posibilitatea de a defini un **set infinit de obiecte** printr-o **relație** sau un **set finit de relații**, caracteristică evidențiată foarte bine de exemplele furnizate mai sus.

- Cunoscută ca și un *concept fundamental* în **matematică**, recursivitatea a devenit și o puternică facilitare de programare odată cu apariția limbajelor de programare de nivel superior care o implementează ca și **caracteristică intrinsecă** (ALGOL, PASCAL, C, JAVA etc.).

- În contextul programării, **recursivitatea** este strâns legată de **iterație** și pentru a nu da naștere unor confuzii, se vor defini în continuare cele două concepte din punctul de vedere al **tehnicilor de programare**.

- **Iterația** este **execuția repetată** a unei porțiuni de program, până în momentul în care se îndeplinește o **condiție** respectiv atâta timp cât condiția este îndeplinită.

- Fiecare execuție se duce până la **capăt**, se verifică îndeplinirea condiției și în caz de răspuns nesatisfăcător se reia execuția de la început.

- Exemple clasice în acest sens sunt **structurile repetitive WHILE, REPEAT și FOR** .

- **Recursivitatea** presupune de asemenea **execuția repetată** a unei porțiuni de program.

- În **contrast** cu iterația însă, în cadrul recursivității, condiția este verificată în **cursul** execuției programului și nu la sfârșitul ei ca la iterație.
- În caz de rezultat nesatisfăcător, întreaga porțiune de program este apelată din nou ca **subprogram (procedură sau funcție) a ei însăși**, în particular **ca procedură a porțiunii de program originale** care încă **nu** și-a terminat execuția.
- În momentul satisfacerii condiției de revenire, **se reia execuția programului apelant** exact din punctul în care s-a apelat pe el însuși.
- Acest lucru este valabil pentru **toate apelurile** anterioare satisfacerii condiției.
- Utilizarea **algoritmilor recursivi** este potrivită în situațiile care presupun recursivitate respectiv calcule, funcții sau structuri de date definite în termeni recursivi.
- În general, un **program recursiv** P, poate fi exprimat formal prin mulțimea P care constă din secvențele de instrucțiuni S_i, S_j care nu îl conțin pe P și P însuși. Se precizează ca fie S_i fie S_j pot fi secvența vidă. [5.1.1.b].

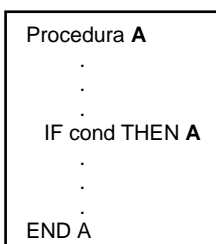
$P \equiv P [S_i, P, S_j]$

[5.1.1.b]

5.1.2. Mecanismul implementării recursivității

- **Structurile de program** necesare și suficiente pentru exprimarea recursivității sunt **procedurile, subrutinele sau funcțiile** care pot fi apelate prin nume.
 - Dacă o procedură P conține o referință directă la ea însăși, se spune că este **direct recursivă**.
 - Dacă P conține o referință la o altă procedură Q care la rândul ei conține o referință (directă sau indirectă) la P, se spune că P este **recursivă indirect** (figura 5.1.2.a).

(a)



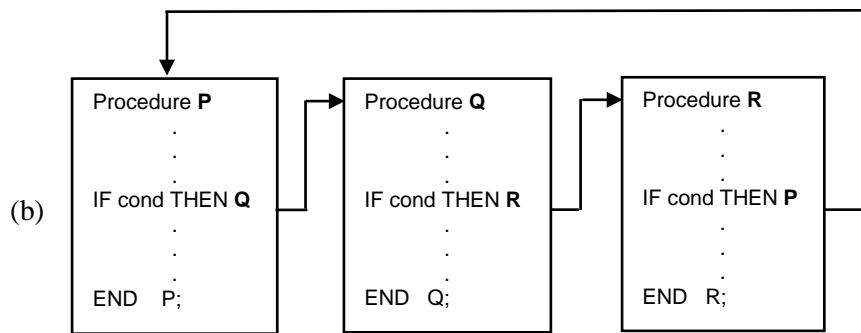


Fig.5.1.2.a. Recursivitate directă (a) și recursivitate indirectă (b)

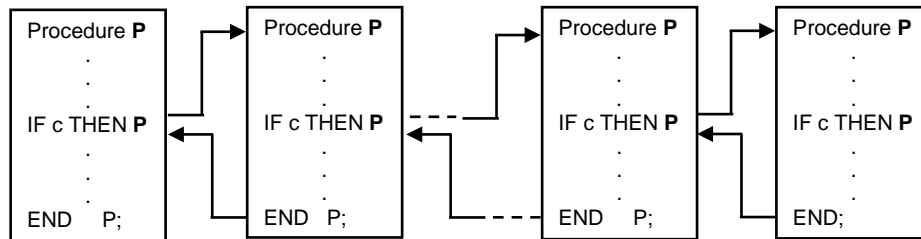


Fig.5.1.2.b. Model de execuție al unei proceduri recursive

- Intuitiv, **execuția unei proceduri recursive** este prezentată în figura 5.1.2.b.
 - Fiecare reprezentare asociată procedurii reprezintă o **instanță de apel recursiv** a acesteia.
 - Instanțele se apelează unele pe altele până în momentul în care condiția *c* devine falsă.
 - În acest moment **se revine pe lanțul apelurilor în ordine inversă**, continuând execuția fiecărei instanțe din punctul de autoapel până la sfârșit, după cum indică săgețile.
 - Suprapunând imaginar toate aceste figuri peste una singură, se obține **modelul de execuție** al unei **proceduri recursive**.
- De regulă, unei **proceduri** *i* se asociază un **set de obiecte locale procedurii** (variabile, constante, tipuri și proceduri) care sunt definite local în procedură și care nu există sau nu au înțeles în afara ei. Acest set de obiecte locale procedurii include și **parametrii instanței de apel**.

- Mulțimea tuturor acestor obiecte constituie așa numitul **context al procedurii**.
- De fiecare dată când o astfel de procedură este apelată recursiv, se creează un **nou** set de astfel de variabile locale specifice apelului cu alte cuvinte un **nou context specific apelului**, care se salvează în **stiva sistem**.
- Deși variabilele din acest context, numit **context curent**, au același **nume** ca și cele corespunzătoare lor din instanța anterioară a procedurii (în calitate de program apelant), ele au valori **distincte** și orice **conflict** de nume este evitat prin **regula** care stabilește **domeniul de existență al identificatorilor**.
 - **Regula** este următoarea:
 - **Identificatorii** se referă întotdeauna la **setul cel mai recent creat de variabile**, adică la **contextul curent**, aflat în **vârful stivei**.
 - Aceași regulă este valabilă și pentru **parametrii de apel** ai procedurii care sunt asociați prin definiție setului de variabile, respectiv contextului apelului.
- Pe măsură ce se realizează apeluri recursive, **contextul** fiecărui apel este **salvat** în **stivă**, iar pe măsură ce execuția unei instanțe s-a încheiat, se **restaurează** contextul instanței apelante prin eliminarea contextului aflat în vârful stivei.
- Acest **mecanism de implementare a recursivității** este ilustrat în figura 5.1.2.c.

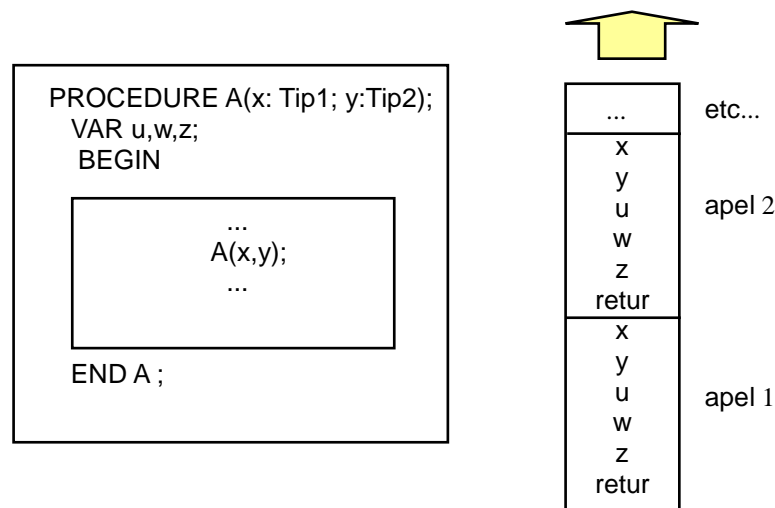


Fig.5.1.2.c. Mecanism pentru implementarea recursivității

- În legătură cu acest **mecanism** se pot face câteva observații:
 - (1) În cadrul apelurilor recursive **nu** apar nici un fel de probleme referitoare la **transmiterea parametrilor** interinstanțe de apel, dacă acești parametri se transmit prin **valoare**, ei fiind salvați în stivă odată cu contextul.

- (2) În cazul parametrilor care **se transmit prin adresă** (de tip **VAR** în Pascal, respectiv pointer în C), pentru a face posibilă transmiterea valorilor acestor parametri între instanțele de apel, este necesară **declararea suplimentară a unor parametri locali**, câte unul pentru fiecare parametru de acest tip, care să asigure retransmiterea valorilor calculate interinstanțe. Acești parametri trebuie reasignați cu valorile inițiale înaintea revenirii din procedura recursivă [5.1.2.a].

```

-----
PROCEDURE A(x: Tip1; VAR y: Tip2; VAR z: Tip3);
  VAR u,w: TipX;
    y1: Tip2; {parametru local pentru y}
    z1: Tip3; {parametru local pentru z}
  BEGIN
    ...
    A(x,y1,z1);
    ...
    y:= y1; {reassignare parametru inițial y}
    z:= z1; {reassignare parametru inițial z}
  END; {A}
-----

void a(tip1 x, tip2* y, tip3* z){
  tipx u,w;
  tip2 y1; /*parametru local pentru y*/
  tip3 z1; /*parametru local pentru z*/
  ...
  a(x,&y1, 0);
  ...
  *y=y1; /*reassignare parametru inițial y*/
  *z=z1; /*reassignare parametru inițial z*/
} /*A*/
/*-----*/

```

- (3) În cazul parametrilor de tip **referință** trebuie ținut cont de faptul că în urma apelului recursiv al unei proceduri se realizează **automat** și **atribuirea** parametrilor implicați în apel [5.1.2.b].

```

-----
PROCEDURE A(r: TipReferinta);
  BEGIN
    ...
    A(r^.drept); {simultan cu apelul se realizează și
                  atribuirea variabilei curente r}
    ...
  END; {A}
-----

void a(tipreferinta r){
  ...
  a(r->drept); /*simultan cu apelul se realizează și
                atribuirea variabilei curente r*/
  ...
} /*A*/
/*-----*/

```

- Ca și în cazul structurilor repetitive, procedurile recursive necesită evaluarea unei **condiții de terminare**, fără de care un apel recursiv conduce la o **buclă de program infinită**.
 - Astfel, orice apel recursiv al procedurii P trebuie supus controlului unei **condiții** c care la un moment dat devine neadevărată.
- În baza acestei observații, un **algoritm recursiv** poate fi exprimat cu mai mare acuratețe prin una din formulele [5.1.2.c], [5.1.2.d] dacă S_i este vidă, respectiv [5.1.2.e] dacă S_j este vidă.

$$P \equiv P [S_i, \text{IF } c \text{ THEN } P, S_j] \quad [5.1.2.c]$$

$$P \equiv P [\text{IF } c \text{ THEN } P, S_j] \quad [5.1.2.d]$$

$$P \equiv P [S_i, \text{IF } c \text{ THEN } P] \quad [5.1.2.e]$$

- **Tehnica de bază** în demonstrarea **terminării** unei **iterații (repetiții)** constă în:
 - (1) A **defini** o funcție $f(x)$ (unde x este o variabilă din program), astfel încât **condiția** $f(x) < 0$ să implice satisfacerea condiției de terminare a iterației.
 - (2) A **demonstra** că $f(x)$ **descrește** pe parcursul execuției iterației respective.
- Într-o manieră similară, **terminarea** unui **algoritm recursiv** poate fi demonstrată, dovedind că P **descrește** pe $f(x)$.
 - O modalitate particulară uzuală de a realiza acest lucru, este de a asocia lui P un **parametru**, spre exemplu n și de a apela recursiv pe P utilizând pe $n-1$ ca și valoare de parametru.
 - Dacă se înlocuiește **condiția** c cu $n > 0$, se garantează **terminarea**.
 - Acest lucru se poate exprima **formal** cu ajutorul următoarei scheme de program [5.1.2.f]:

$$P(n) \equiv P [S_i, \text{IF } n > 0 \text{ THEN } P(n-1), S_j] \quad [5.1.2.f]$$

- De regulă în aplicațiile practice trebuie demonstrat **nu** numai că **adâncimea recursivității este finită** ci și faptul că ea este **suficient de mică**.
 - Motivul este că fiecare apel recursiv al procedurii P , necesită **alocarea** unui volum de memorie variabilelor sale curente (contextului procedurii).
 - În plus, alături de aceste variabile trebuie memorată și **starea curentă** a programului, cu scopul de a fi **refăcută**, atunci când apelul curent al lui P se termină și urmează să fie reluată instanța apelantă.

- Neglijarea acestor aspecte poate avea drept consecință **depășirea** spațiului de memorie alocat stivei sistem, greșeală frecventă în contextul utilizării procedurilor recursive.

5.1.3. Exemple de programe recursive

- Pentru clarificarea conceptelor mai sus prezentate se furnizează în continuare trei exemple de **programe recursive simple**.

5.1.3.1. Algoritm recursiv simplu

- Scopul programului furnizat în continuare ca exemplu, este acela de a ilustra **principiul de funcționare** al unei proceduri recursive.
- În cadrul programului se definește procedura recursivă **Revers** :
 - Procedura **Revers** citește câte un caracter și îl afișează.
 - În acest scop în cadrul procedurii se declară variabila locală `z : char`.
 - Procedura verifică dacă caracterul citit este blank:
 - În caz **negativ** procedura se autoapelează recursiv.
 - În caz **afirmativ** afișează caracterul `z` [5.1.3.1.a].

{Exemplu de program recursiv simplu - varianta Pascal}

```
PROCEDURE Revers;
  VAR z: char;
  BEGIN
    Read(z);
    IF z<>' ' THEN Revers;
    Write(z)
  END; {Revers}
```

[5.1.3.1.a]

/* Exemplu de program recursiv simplu - varianta C */

```
void revers(){
    char z;
    scanf("%c", &z);
    if (z != " ")
        revers();
    printf("%c", z);
}
/*-----*/
```

/*[5.1.3.1.a]*/

- Execuția procedurii **Revers** conduce inițial la afișarea caracterelor în ordinea în care sunt citite până la apariția primului blank.

- Fiecare autoapel presupune salvarea în stiva sistemului a **contextului apelului** care în situația de față constă doar din variabila locală *z*.
- Apariția primului caracter blanc presupune suprimarea apelurilor recursive ale procedurii declanșând continuarea execuției procedurii, până la terminarea sa pentru fiecare din apelurile anterioare.
 - În cazul de față, aceasta presupune afișarea caracterului memorat în variabila *z*.
- Acest șir de reveniri va produce la început afișarea unui blanc, după care sunt afișate caracterele în **ordinea inversă** citirii lor.
 - Acest lucru se întâmplă deoarece fiecare **terminare** a execuției procedurii determină revenirea în apelul anterior, revenire care presupune **reactualizarea contextului apelului**.
 - Întrucât contextele sunt salvate în **stivă**, reactualizarea lor se face **în sens invers** ordinii în care au fost memorate.
- De fapt pentru fiecare cuvânt introdus procedura *Revers* furnizează cuvântul în cauză urmat de același cuvânt **scris invers**.

5.1.3.2. Algoritm pentru traversare a unei liste înlănțuite

- Exemplul următor prezintă un **algoritm recursiv** pentru **traversarea unei liste înlănțuite** [5.1.3.2.a].

{Traversarea recursivă a unei liste înlănțuite - varianta Pascal}

```

TYPE TipLista = ^TipNod;
      TipNod = RECORD
        data: TipData;
        urm: TipLista
      END;
  
```

```

PROCEDURE Traversare(p:TipLista);                                [5.1.3.2.a]
BEGIN
  IF p<>nil THEN
    BEGIN
[1]      Prelucrare(p^.data);
[2]      Traversare(p^.urm)
    END
  END; {Traversare}
  
```

/* Traversarea recursivă a unei liste înlănțuite - varianta C */

```

typedef struct tipnod* tiplista;
  
```

```

typedef struct {
    int data;
    tiplista urm;
} tipnod;

void traversare(tiplista p)                                /*[5.1.3.2.a]*/
{
    if (p!=0){
/*[1]*/      prelucrare(p->data);
/*[2]*/      traversare(p->urm);
    }
} /*Traversare*/
/*-----*/

```

- Algoritmul de traversare **nu** numai că este foarte **simplu** și foarte **elegant**, dar prin simpla **inversare** a instrucțiunilor Prelucrare și Traversare ([1]cu [2]) se obține parcurgerea în sens **invers** a listei în cauză.
- În acest ultim caz comportamentul algoritmului se poate descrie astfel:
 - Se traversează mai întâi lista începând cu poziția curentă până la sfârșitul listei.
 - După ce s-a realizat această traversare se prelucrează informația din nodul curent.
- **Consecința** execuției algoritmului:
 - Informația conținută într-o anumită poziție p este prelucrată **numai** după ce au fost prelucrate **toate** informațiile corespunzătoare tuturor nodurilor care îi urmează lui p.

5.1.3.3. Algoritm pentru rezolvarea problemei Turnurilor din Hanoi

- Exemplul următor se referă la binecunoscuta problemă a **Turnurilor din Hanoi** a cărei **specificare** este următoarea:
 - Se consideră trei **vergele** A, B și C.
 - Se consideră de asemenea un set de n **discuri găurite** fiecare de altă dimensiune, care sunt amplasate în ordine descrescătoare, de jos în sus pe vergeaua A (figura 5.1.3.3.a).

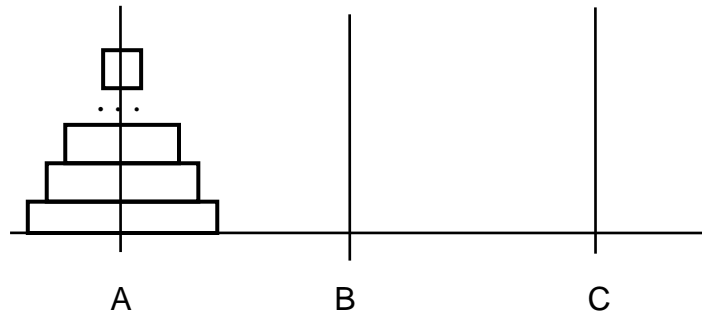


Fig.5.1.3.3.a. Model pentru problema Turnurilor din Hanoi

- Se cere să se **mute** discurile pe vergeaua C utilizând ca și auxiliar vergeaua B.
- Se va respecta următoarea **restricție**:
 - **Nu** se așează niciodată un disc mai mare peste un disc mai mic.
- Problema pare simplă, dar rezolvarea ei cere multă, răbdare, acuratețe și un volum de timp care crește exponențial odată cu n.
 - O abordare **recursivă** însă reprezintă o soluție simplă și elegantă.
- **Rezolvarea recursivă** a problemei presupune abordarea unui **caz simplu**, urmată de **generalizarea** corespunzătoare.
 - Pentru început se consideră că există doar **două discuri** numerotate cu 1 (cel mai mic situat deasupra) respectiv cu 2 (cel mai mare), a căror mutare în condițiile impuse presupune următorii pași :
 - (1) Se mută discul 1 de pe A pe B.
 - (2) Se mută discul 2 de pe A pe C.
 - (3) Se mută discul 1 de pe B pe C.
 - Prin **generalizare** se reia același algoritm pentru n discuri (figura 5.1.3.3.b), adică:
 - (1) Se mută n-1 discuri (cele de deasupra) de pe A pe B, prin C.
 - (2) Se mută discul n de pe A pe C.
 - (3) Se mută n-1 discuri de pe B pe C, prin A.

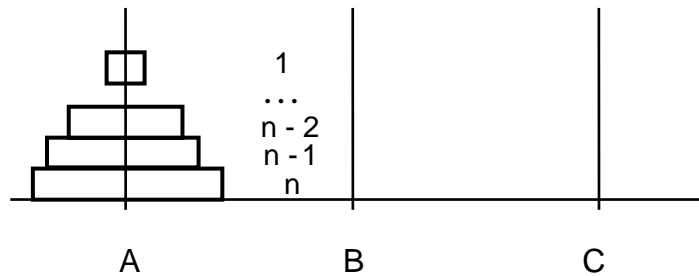


Fig.5.1.3.3.b. Model generalizat al problemei Turnurilor din Hanoi

- Pornind de la acest **model**, o variantă imediată de **implementare recursivă** a rezolvării problemei este prezentată în secvența [5.1.3.3.a].

{Turnurile din Hanoi - varianta Pascal}

```

PROCEDURE TurnuriHanoi(NrDiscuri:integer; A{dela},
                        B{la}, C{prin}: TipVergele);
  PROCEDURE MutaDisc(x{de la},y{la}: TipVergele);
    BEGIN
      *muta discul de la x la y
    END; {MutaDisc}
  BEGIN {TurnuriHanoi}
    IF NrDiscuri=1 THEN
      MutaDisc(A,C)
    ELSE
      BEGIN
        TurnuriHanoi(NrDiscuri-1, A, B, C);
        MutaDisc(A,C);
        TurnuriHanoi(NrDiscuri-1, B, C, A)
      END {ELSE}
    END; {TurnuriHanoi}

```

[5.1.3.3.a]

/* Turnurile din Hanoi - varianta C */

```

void muta_disc(tip_vergea x/*de la*/,tip_vergea y/*la*/)
{
  /* muta discul de la x la y*/
}

void turnuri_hanoi(int nr_discuri, tip_vergea a/*dela*/,
                  tip_vergea b/*la*/,
                  tip_vergea c/*prin*/)

{
  /*turnuri_hanoi*/
  if (nr_discuri==1)
    muta_disc(a,c);
  else
  {
    turnuri_hanoi(nr_discuri-1, a, b, c);
    muta_disc(a,c);
    turnuri_hanoi(nr_discuri-1, b, c, a);
  }
}

```

/*[5.1.3.3a]*/

```

    }      /*else*/
}      /*turnuri_hanoi*/

/*-----*/

```

- Procedura **MutaDisc** realizează efectiv mutarea unui disc de pe o vergea sursă pe o vergea destinație.
- Procedura **TurnuriHanoi** realizează două auto-apeluri recursive și un apel al procedurii **MutaDisc** conform modelului generalizat prezentat mai sus.

5.2. Utilizarea recursivității

5.2.1. Cazul general de utilizare a recursivității

- **Algoritmii recursivi** sunt potriviți a fi utilizați atunci când **problema** care trebuie rezolvată sau **datele** care trebuiesc prelucrate **sunt definite în termeni recursivi**.
- Cu toate acestea, un astfel de mod de definire **nu** justifică întotdeauna faptul că utilizarea unui algoritm recursiv reprezintă cea mai bună alegere.
 - Mai mult, utilizarea recursivității în anumite situații nepotrivite, coroborată cu regia relativ ridicată a implementării și execuției unor astfel de algoritmi, a generat în timp un curent de opinie **potrivnic** destul de vehement.
- Cu toate acestea recursivitatea rămâne o **tehnică de programare fundamentală** cu un domeniu de aplicabilitate foarte bine delimitat.
- În continuare se prezintă un exemplu de construcție a unui program recursiv pornind de la **modelul generic recursiv** [5.2.1.a] derivat din modelul [5.1.2.d].

$$P \equiv \text{IF } c \text{ THEN } P[S_i, P] \quad [5.2.1.a]$$

- Acest model este foarte potrivit în cazurile în care se cere calculul unor valori care se definesc cu ajutorul unor **relații recurente**.
- Un exemplu clasic în acest sens îl reprezintă **numerele factoriale** $f_i = i!$ precizate în [5.2.1.b].

$$\begin{aligned} i &= 0, 1, 2, 3, 4, 5, \dots \\ f_i &= 1, 1, 2, 6, 24, 120, \dots \end{aligned} \quad [5.2.1.b]$$

- Elementul cu indicele 0 este furnizat în mod explicit ca fiind egal cu 1.
- Elementele următoare, sunt definite în mod recursiv, în termenii predecesorilor lor [5.2.1.c].

$$f_{i+1} = (i+1) * f_i \quad [5.2.1.c]$$

- Pornind de la această formulă, se introduc două variabile i și f care precizează respectiv valorile lui i și f_i , în cel de-al i -lea nivel de recursivitate rezultând următoarea secvență S_i [5.2.1.d].

$$S_i \equiv (i := i+1; f := i*f;) \quad [5.2.1.d]$$

- Înlocuind pe S_i din [5.2.1.d] în [5.2.1.a] se obține **forma recursivă** [5.2.1.e] căreia i se poate asocia programul principal [5.2.1.f].

$$P \equiv \text{IF } i < n \text{ THEN } (i := i+1; f := i*f; P) \quad [5.2.1.e]$$

$$i := 0; \quad f := 1; \quad P; \quad [5.2.1.f]$$

- Varianta imediată de implementare a procedurii P apare în secvența [5.2.1.g].

{Calculul factorialului - Varianta 1 Pascal}

```
PROCEDURE P;
  BEGIN
    IF i < n THEN
      BEGIN
        i := i+1; f := i*f; P
      END
    END; {P}
```

/* Calculul factorialului - Varianta 1 C */

```
void p()
{
  if (i < n) {
    i++;
    f *= i;
    p();
  }
} /*p*/
/*-----*/
```

- După cum s-a precizat, algoritmii recursivi se recomandă a fi utilizați cu precădere pentru implementarea unor probleme care se pot defini în **manieră recursivă**.
 - Cu toate acestea, **recursivitatea** poate fi utilizată și în cazul unor probleme **de natură nerecursivă**.
- Spre exemplu, în [5.2.1.h] se prezintă un exemplu de **implementare recursivă** a unei căutări într-un tablou liniar a .

{Cautare recursivă într-un tablou liniar - varianta Pascal}

```
VAR n: integer; x: TipElement;
    a: ARRAY[1..n] OF TipElement;
```

```

FUNCTION Cauta(i: TipIndice): TipIndice;
BEGIN
    IF i>n THEN Cauta:= 0 ELSE                                [5.2.1.h]
        IF a[i]=x THEN Cauta:= i ELSE
            Cauta:= Cauta(i+1)
    END;{Cauta}
{Apelul inițial: Cauta(1)}
-----
/* Cautare recursivă într-un tablou liniar - varianta C */

int n; tip_element x;
tip_element *a;

tip_indice cauta(tip_indice i){
    tip_indice cauta_result;
    if (i>n)
        cauta_result=0;
    else                                /*[5.2.1.h]*/
        if (a[i-1]==x)
            cauta_result=i;
        else
            cauta_result=cauta(i+1);
    return cauta_result;
}    /*cauta*/
/*apelul inițial: cauta(1)*/
/*-----*/

```

- Este evident faptul că o astfel de abordare este posibilă, dar ea este **forțată** și **ineficientă**, varianta iterativă fiind preferată în acest caz.

5.2.2. Algoritm recursiv pentru calculul factorialului

- Procedura P de **calcul a valorii factorialului** [5.2.1.g] prezentată în paragraful anterior este de regulă înlocuită cu un subprogram de tip funcție, din următoarele rațiuni:
 - **Funcția** poate fi utilizată direct ca și constituent al unei expresii.
 - **Funcției** i se poate asocia în mod explicit valoarea rezultată din calcule.
- Variabila f din procedura P devine superfluă, iar rolului lui i este preluat de către parametrul n de apel al funcției [5.2.2.a].

```

-----
{Funcție recursivă pentru calculul factorialului - varianta Pascal}

```

```

FUNCTION Fact(n: integer): integer;
BEGIN
    IF n=0 THEN
        Fact:= 1                                [5.2.2.a]

```

```

        ELSE
            Fact:= n*Fact(n-1)
        END; {Fact}
    -----
    /* Funcție recursivă pentru calculul factorialului -
    varianta C */

    int fact(int n)
    {
        int fact_result;
        if (n==0)
            fact_result=1;                                /*[5.2.2.a]*/
        else
            fact_result=n*fact(n-1);
        return fact_result;
    } /*fact*/
    /*-----*/

```

- Secvența în cauză care a fost redactată pentru calculul factorialului ca și **întreg**, este **limitată** din punctul de vedere al dimensiunii maxime a lui n din rațiuni de reprezentare în calculator a numerelor întregi.
 - Trecerea la varianta **reală**, eliberată de această constrângere se realizează simplu.
- În figura 5.2.2.a apare reprezentarea intuitivă a **apelurilor recursive** ale funcției **Fact** pentru $n = 4$.

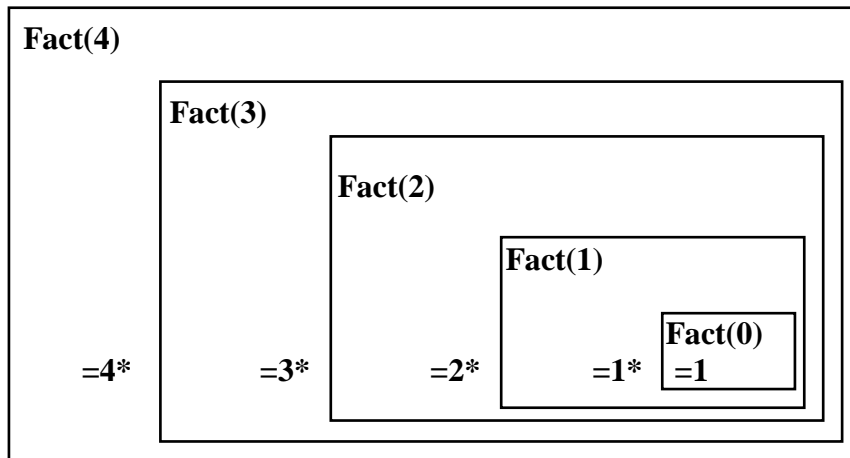


Fig.5.2.2.a. Apelurile ale funcției recursive **Fact** pentru $n = 4$

- În figura 5.2.2.b, se prezintă intuitiv **revenirile** succesive din apelurile recursive ale funcției **Fact**, care au drept urmare calculul efectiv al valorii factorialului.

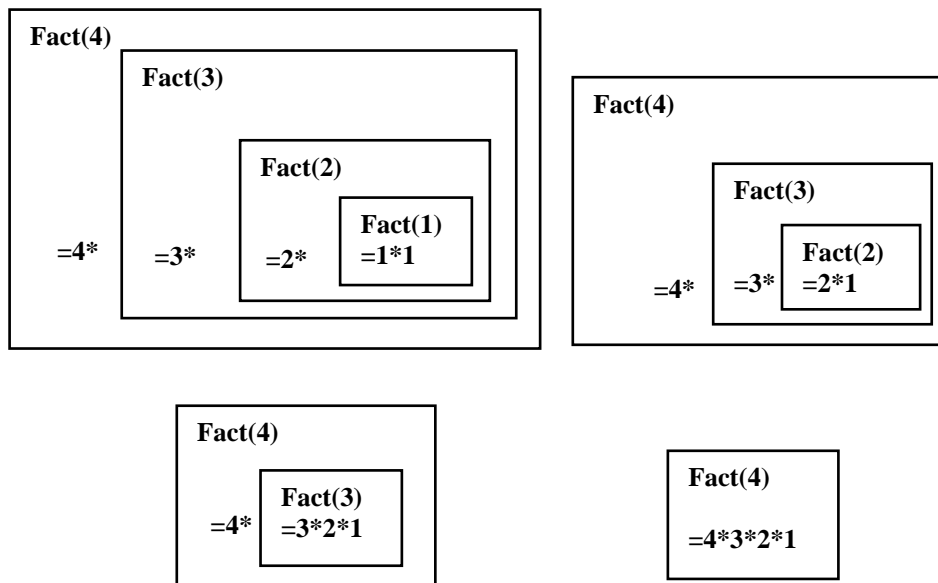


Fig.5.2.2.b. Rezolvarea apelurilor funcției recursive `Fact` pentru $n = 4$

- În figura 5.2.2.c apare **structura apelurilor** respectiv a rezolvării acestora în formă de **arbore de apeluri**.

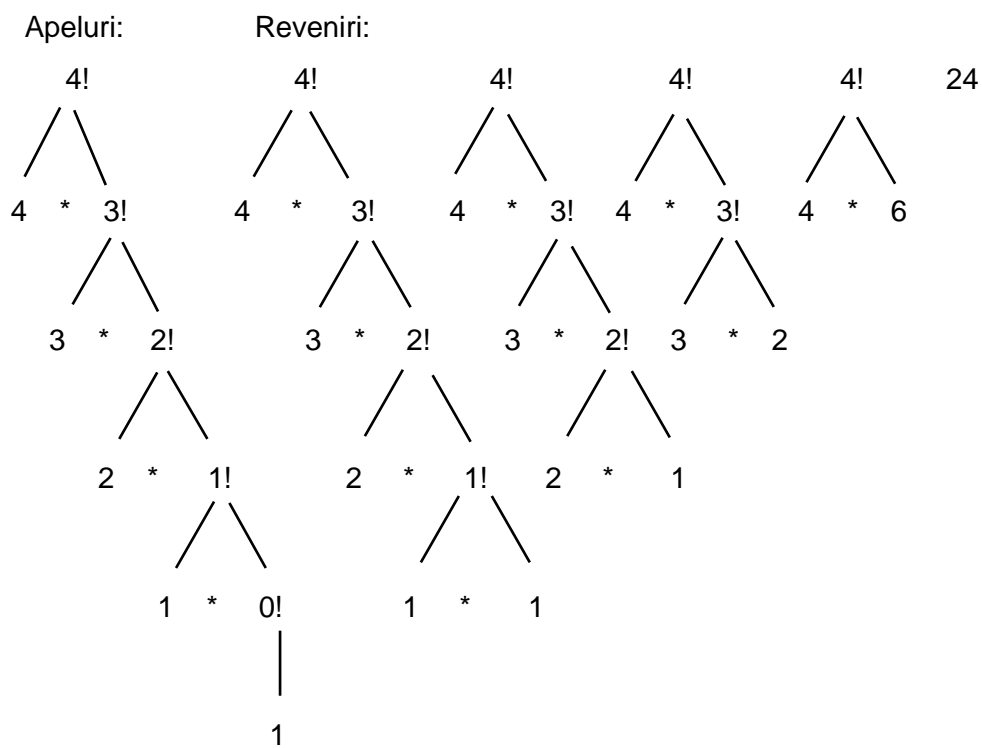


Fig.5.2.2.c. Arborele de apeluri al funcției `Fact (4)`

- În acest caz fiind vorba despre o funcție cu **un singur apel recursiv**, **arborele de apeluri** are o structură de **listă liniară**.
 - Fiecare apel adaugă un element la sfârșitul acestei liste, iar rezolvarea apelurilor are drept consecință reducerea succesivă a listei de la sfârșit spre început.
- După cum se observă, pentru a calcula factorialul de ordinul n sunt necesare $n+1$ apeluri ale funcției recursive **Fact**.
- Este însă evident că în cazul calculului factorialului, recursivitatea poate fi înlocuită printr-o simplă **iterație** [5.2.2.b].

{Calculul factorialului - implementare iterativă - Varianta Pascal}

```
VAR i,fact: integer;
i:= 0; fact:= 1;
WHILE i<n DO                                [ 5.2.2.b]
  BEGIN
    i:= i+1; fact:= fact*i
  END;
```

/* Calculul factorialului - implementare iterativă - Varianta C */

```
int i,fact;
i=0;
fact=1;
while (i<n)                                /*[ 5.2.2.b]*/
{
  i++; fact*=i;
}
/*-----*/
```

- În figura 5.2.2.d. apare **reprezentarea grafică a profilului performanței** algoritmului de **calcul al factorialului** în variantă **recursivă** respectiv **iterativă**.
 - Sunt de fapt reprezentați în manieră comparativă, **timpii de execuție** pe un același sistem de calcul, ai celor doi algoritmi funcție de valoarea lui n .
 - După cum se observă, deși **ambii algoritmi** sunt **liniari** în raport cu n , adică au performanța $O(n)$, algoritmul recursiv este de aproximativ 4 ori mai lent decât cel iterativ.
 - Expresiile analitice ale celor două reprezentări apar în [5.2.2.c][De84].

$T_i(n)=0.0018*n+0.0033$ [5.2.2.c]
 $T_r(n)=0.0056*n+0.017$

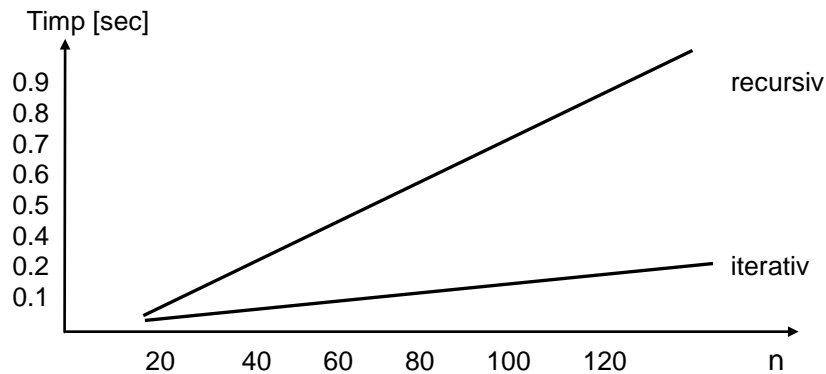


Fig.5.2.2.d. Profilul algoritmului factorial (variantele recursivă și iterativă)

5.2.3. Calculul numerelor lui Fibonacci

- Există și alte exemple de definiții recursive care se tratează mult mai eficient cu ajutorul iterației.
- Un exemplu în acest sens îl reprezintă **calculul numerelor lui Fibonacci de ordin 1** care sunt definite prin următoarea relație recursivă [5.2.3.a]:

$$\begin{aligned} \text{Fib}_{n+1} &= \text{Fib}_n + \text{Fib}_{n-1} && \text{pentru } n > 0 \\ \text{Fib}_1 &= 1; \text{Fib}_0 = 0 && [5.2.3.a] \end{aligned}$$

- Această definiție recursivă conduce imediat la următorul algoritm de calcul recursiv [5.2.3.b].

{Calculul numerelor lui Fibonacci - Varianta recursivă - Pascal}

FUNCTION Fib(n: integer): integer;

BEGIN

IF n=0 **THEN** Fib:= 0 **ELSE** [5.2.3.b]

IF n=1 **THEN** Fib:= 1 **ELSE**

Fib:= Fib(n-1) + Fib(n-2)

END;{Fib}

/* Calculul numerelor lui Fibonacci - Varianta recursivă - C */

int fib(int n)

{

int fib_result;

if (n==0) fib_result=0; else /*[5.2.3.b]*/

if (n==1) fib_result=1; else

fib_result=fib(n-1) + fib(n-2);

return fib_result;

} /*fib*/

/*-----*/

- Din păcate modelul recursiv utilizat în această situație conduce la o manieră ineficientă de calcul, deoarece numărul de apeluri crește **exponențial** cu n .
- În figura 5.2.3.a. apare reprezentarea arborelui de apeluri pentru $n=5$.

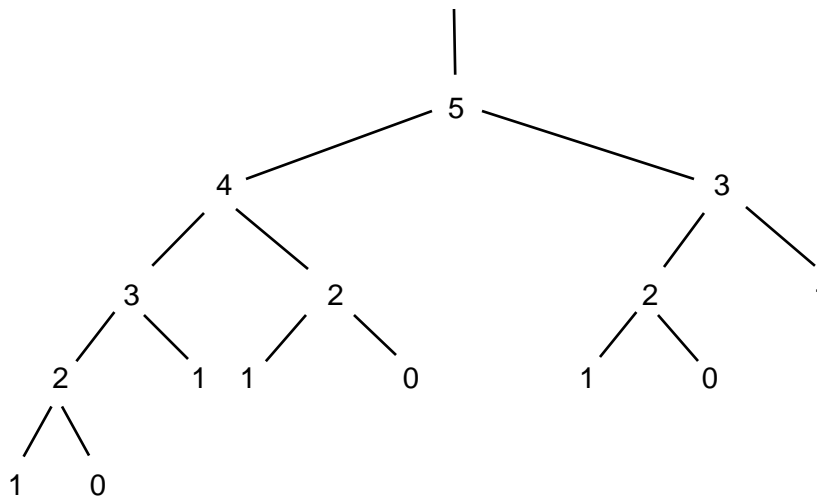


Fig.5.2.3.a. Arborele de apeluri al procedurii `Fib` pentru $n = 5$

- După cum se observă:
 - Este vorba despre un **arbore binar de apeluri** întrucât există două apeluri recursive ale procedurii **Fib**.
 - Parcurgerea arborelui de apeluri necesită 15 apeluri, fiecare nod semnificând un apel al procedurii.
 - Ineficiența acestei implementări crește odată cu creșterea lui n .
- Este evident faptul că **numerele lui Fibonacci** pot fi calculate cu ajutorul unei **scheme iterative** care elimină recalcularea valorilor, utilizând două variabile auxiliare $x = \text{Fib}_i$ și $y = \text{Fib}_{i-1}$ [5.2.3.c].

{Calculul numerelor lui Fibonacci - Varianta iterativă - Pascal}

```
i:= 1; x:= 1; y:= 0;
```

```
WHILE i<n DO
```

[5.2.3.c]

```
  BEGIN
```

```
    z:= x; i:= i+1;
```

```
    x:= x+y; y:= z
```

```
  END;
```

```
/* Calculul numerelor lui Fibonacci - Varianta iterativă - C */
```

```

i=1; x=1; y=0;
while (i<n)                                     /*[5.2.3.c]*/
{
    z=x; i++;
    x=x+y; y=z;
}
/*-----*/

```

- Variabila auxiliară z poate fi evitată, utilizând atribuiri de forma $x := x+y$ și $y := x-y$.

5.2.4. Eliminarea recursivității

- **Recursivitatea** reprezintă o **facilitate** excelentă de programare care contribuie la exprimarea simplă, concisă și elegantă a algoritmilor de natură recursivă.
 - Cu toate acestea, ori de câte ori problemele eficienței și performanței se pun cu preponderență, se recomandă **evitarea utilizării recursivității**.
 - De asemenea se recomandă evitarea recursivității ori de câte ori stă la dispoziție o **rezolvare evidentă** bazată pe **iterație**.
- De fapt, implementarea recursivității pe echipamente nerecursive dovedește faptul că **orice** algoritm **recursiv** poate fi transformat într-unul **iterativ**.
- În continuare se abordează teoretic **problema conversiei** unui **algoritm recursiv** într-unul **iterativ**.
- În abordarea acestei chestiuni se disting **două cazuri**:
- (1) Cazul în care **apelul recursiv** al procedurii apare la **sfârșitul** ei, drept ultimă instrucțiune a procedurii ("**tail recursion**").
 - În această situație, **recursivitatea** poate fi înlocuită cu o **buclă** simplă de **iterație**.
 - Acest lucru este posibil, deoarece în acest caz, **revenirea** dintr-un apel încuibat presupune și **terminarea** instanței respective a procedurii, motiv pentru care contextul apelului **nu** mai trebuie restaurat.
 - Astfel dacă o procedură $P(x)$ conține ca și **ultim pas** al său un apel la ea însăși de forma $P(y)$:
 - Acest apel poate fi înlocuit cu o instrucțiune de atribuire $x := y$, urmată de reluarea (salt la începutul) codului lui P .
 - y poate fi și o expresie.

- x trebuie să fie în mod obligatoriu o variabilă transmisibilă prin **valoare**, astfel încât valoarea ei să fie memorată într-o locație specifică apelului.
- x poate fi transmis prin **referință** dacă y este chiar x .
- Dacă P are mai mulți parametri, ei pot fi tratați fiecare în parte ca x și y .
- Această modificare este valabilă deoarece reluarea execuției lui P , cu noua valoare a lui x , are exact același efect ca și când s-ar apela $P(y)$ și s-ar reveni din acest apel.
- În general **programele recursive** de acest tip pot fi convertite formal în **forme iterative** după cum urmează.
 - Forma **recursivă** [5.2.4.a] devine forma **iterativă** [5.2.4.b] iar [5.2.4.c] devine [5.2.4.d].

$P(x) \equiv P [S_i, \text{IF } c \text{ THEN } P(y)]$ [5.2.4.a]

$P(x) \equiv P [S_i, \text{IF } c \text{ THEN } [x:=y; \text{reluare } P]]$ [5.2.4.b]

$P(n) \equiv P [S_i, \text{IF } n>0 \text{ THEN } P(n-1)]$ [5.2.4.c]

$P(n) \equiv P [S_i, \text{IF } n>0 \text{ THEN } [n:=n-1; \text{reluare } P]]$ [5.2.4.d]

- Exemple concludente în acest sens sunt implementarea iterativă a **calculului factorialului** și a **calculului șirului numerelor lui Fibonacci** prezentate anterior.
- (2) Cazul în care **apelul** sau **apelurile recursive** se realizează din **interiorul procedurii** [5.2.4.e].

$P \equiv P [S_i, \text{IF } c \text{ THEN } P, S_j]$ [5.2.4.e]

- Varianta iterativă a acestei situații presupune tratarea explicită de către programator a **stivei apelurilor**.

- Fiecare **apel** necesită **salvarea** în **stiva** gestionată de către utilizator a **contextului instanței de apel**.
- Acest context trebuie restaurat de către utilizator din aceeași stivă la terminarea instanței și revenirea în instanța anterioară de apel.
- Activitatea de conversie în acest caz are un **înalt grad de specificitate** funcție de algoritmul recursiv care se dorește a fi convertit, este în general mai dificilă, mai laborioasă și îngreunează înțelegerea algoritmului.
 - Un exemplu în acest sens îl reprezintă **partiționarea nerecursivă** prezentată în paragraful &3.2.6.
- Recursivitatea are însă domeniile ei bine definite în care se aplică cu succes.
- În general se apreciază că algoritmi a căror **natură** este **recursivă** este indicat a fi formulați ca și **proceduri recursive**, lucru pus în evidență de algoritmi prezentați în cadrul acestui capitol și în capitolele următoare.

5.3. Exemple de algoritmi recursivi

5.3.1. Algoritmi care implementează definiții recursive

- În continuare se prezintă două exemple de algoritmi care implementează definiții recursive. Este vorba despre algoritmul lui Euclid pentru determinarea celui mai mare divizor comun a două numere întregi date, respectiv algoritmul de transformare a unei expresii aritmetice din format infix în format postfix.

5.3.1.1. Algoritmul lui Euclid

- **Algoritmul lui Euclid.** Permite determinarea **celui mai mare divizor comun (c.m.m.d.c)** a două numere întregi date.
 - Se definește în manieră **recursivă** după cum urmează:
 - Dacă unul dintre numere este nul, c.m.m.d.c. al lor este celălalt număr.
 - Dacă nici unul din numere **nu** este nul, atunci c.m.m.d.c **nu** se modifică dacă se înlocuiește unul din numere cu restul împărțirii sale cu celălalt.
- Pornind de la această definiție recursivă se poate concepe un subprogram simplu de tip funcție pentru calculul c.m.m.d.c. a două numere întregi [5.3.1.1.a].

{Implementarea algoritmului lui Euclid - varianta Pascal}

```

FUNCTION Cmmdc(m,n: integer): integer;
BEGIN
  IF n=0 THEN
    Cmmdc := m
  ELSE

```

[5 . 3 . 1 . 1 . a]

```

        Cmmdc:= Cmmdc(n, m MOD n)
    END; {Cmmdc}
-----
/* Implementarea algoritmului lui Euclid - varianta C */

int cmmdc(int m,int n)
{
    int cmmdc_result;
    if (n==0)
        cmmdc_result=m;                      /*[5.3.1.1.a]*/
    else
        cmmdc_result=cmmdc(n, m%n);
    return cmmdc_result;
} /*Cmmdc*/
/*-----*/

```

- În figura 5.3.1.1.a apare urma execuției acestui algoritm pentru valorile m=18 și n=27.

m	n	Cmmdc(n, m MOD n)
18	27	Cmmdc(27, 18 mod 27)
27	18	Cmmdc(18, 27 mod 18)
18	9	Cmmdc(9, 18 mod 9)
9	0	n = 0 ; Cmmdc = 9

Fig.5.3.1.1.a. Urma execuției algoritmului lui Euclid

5.3.1.2. Algoritm recursiv pentru transformarea expresiilor aritmetice specificate în format infix în format postfix (notație poloneză).

- Expresiile aritmetice uzuale (format **infix**) se definesc în mod **recursiv** cu ajutorul **diagramelor sintactice** prezentate în figura 5.3.1.2.a.
 - **Recursivitatea** de natură **indirectă** a acestei definiții constă în faptul că definiția lui Factor include din nou definiția lui Expresie.
- Se pune **problema** ca pornind de la o **expresie aritmetică corectă** furnizată în format normal (**infix**) să se obțină forma poloneză (**postfix**) a acesteia.
- Se precizează că **notația poloneză** presupune reprezentarea unei operații aritmetice simple în forma "*operand operand operator*" și nu în forma "*operand operator operand*" presupusă de notația infix.

- Spre exemplu $a+b$ devine $ab+$ iar $a+b*(c-d)$ devine $abcd-*+.$
- **Avantajul** notației poloneze este acela că indică **ordinea corectă** a execuției operațiilor **fără** a utiliza paranteze, iar expresia poate fi evaluată printr-o simplă **baleere secvențială** de la stânga la dreapta.

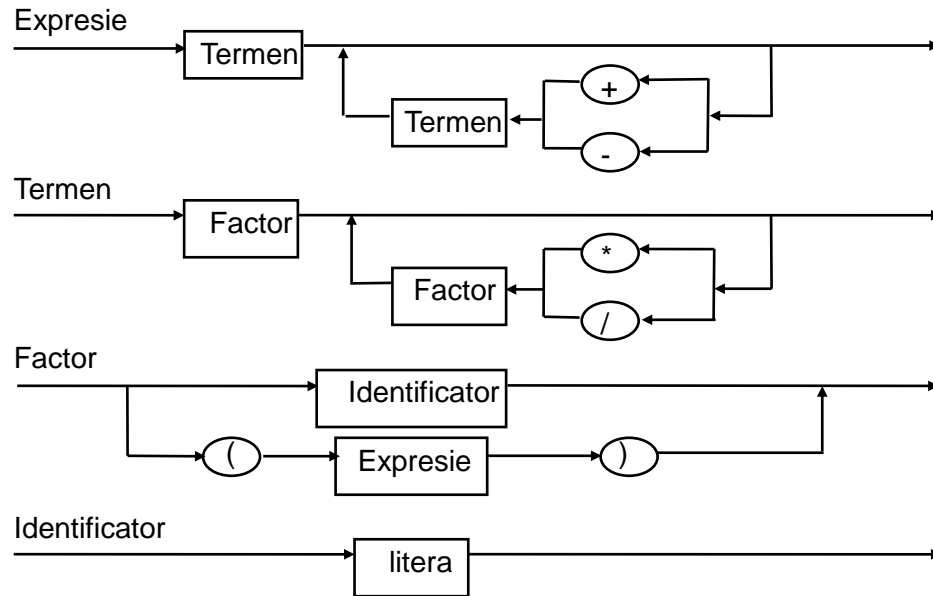


Fig.5.3.1.2.a. Definirea recursivă a unei expresii aritmetice

- **Problema** care trebuie rezolvată este aceea de a **depista** cei **doi operanzi** (care pot fi complecși), de a-i înregistra și de a trece operatorul după ei.
 - Din acest motiv, până la depistarea celui de-al doilea operand, **operatorul aditiv** din cadrul **expresiei** respectiv **operatorul multiplicativ** din cadrul unui **termen**, se **memorează** în variabilele locale `op` respectiv `op1`.
 - Transformarea cerută se poate realiza construind câte o **procedură** individuală de conversie pentru fiecare construcție sintactică (**Expresie**, **Termen**, **Factor**).
 - Deoarece fiecare dintre aceste construcții sintactice este definită recursiv, procedurile corespunzătoare trebuiesc de asemenea activate recursiv.
 - Algoritmul de transformare apare în secvența [5.3.1.2.a].

{Transformarea unei expresii aritmetice din format infix în format postfix - varianta Pascal}

```

TYPE linie = string[80];
VAR infix,post: linie;
    c: char;

```

```

i: integer; {contor expresie curenta}
j: integer; {contor postfix curent}

PROCEDURE CitesteCar(ex: linie; VAR ch: char);
{furnizează caracterul următor al expresiei infix,
eliminând blancurile}
BEGIN
  REPEAT
    ch:= ex[i]; i:= i+1;
  UNTIL ch<>' '
END; {CitesteCar}

PROCEDURE ScrieCar(VAR ex: linie, ch: char);
{depune caracterul în zona precizată}
BEGIN
  ex[j]:= ch; j:= j+1
END; {ScrieCar} [5.3.1.2.a]

PROCEDURE Expresie;
VAR op: char;
PROCEDURE Termen;
VAR op1: char;
PROCEDURE Factor
BEGIN {procedura Factor}
  IF c ='(' THEN
    BEGIN
      CitesteCar(infix,c); Expresie
      {când se revine din expresie ch=')'}
    END
  ELSE
    BEGIN
      ScrieCar(post,c)
    END;
  CitesteCar(infix,c);
END; {Factor}
BEGIN {procedura Termen}
  Factor;
  WHILE (c ='*') OR (c = '/') DO
    BEGIN
      op1:= c; CitesteCar(infix,c);
      Factor;
      ScrieCar(post,op1)
    END {WHILE}
  END; {Termen}
BEGIN {procedura Expresie}
  Termen;
  WHILE (c ='+') OR (c = '-') DO
    BEGIN
      op:= c; CitesteCar(infix,c);
      Termen;
      ScrieCar(post,op)
    END {WHILE}
  END; {Expresie}

```

```
/* Transformarea unei expresii aritmetice din format infix  
în format postfix - varianta C */
```

```
typedef char* linie;
```

```
linie infix,post;
```

```
char c;
```

```
int i;          /*contor expresie curentă*/
```

```
int j;          /*contor postfix curent*/
```

```
void expresie();
```

```
static void termen();
```

```
void citeste_car(linie ex, char* ch)
```

```
    /*furnizează caracterul urmator al expresiei infix,  
    eliminând blancurile*/
```

```
{  
    do {  
        *ch=ex[i]; i++;  
    } while (!( *ch!=' '));  
}    /*citeste_car*/
```

```
void scrie_car(linie* ex, char ch)
```

```
    /*depune caracterul în zona precizată*/
```

```
{  
    (*ex)[j]=ch; j=j+1;  
}    /*scrie_car*/                                     /*[5.3.1.2.a]*/
```

```
static void factor()
```

```
{  
    if (c == '('){  
        citeste_car(infix,&c);  
        expresie();    /*când se revine din expresie ch=')'*/  
    }  
    else{  
        scrie_car(&post,c);  
    }  
    Citeste_car(infix,&c);  
}    /*factor*/
```

```
static void termen()
```

```
{  
    char op1;  
    factor();  
    while ((c == '*' ) || (c == '/'))  
    {  
        op1=c; citeste_car(infix,&c);  
        factor();  
        scrie_car(&post,op1);  
    }    /*while*/  
}    /*termen*/
```

```
void expresie()
```

```
{  
    char op;  
    termen();
```

```

while ((c == '+' ) || (c == '-' ))
{
    op=c;   citeste_car(infix,&c);
    termen();
    scrie_car(&post,op);
} /*while*/
} /*expresie*/
/*-----*/

```

- În cadrul **algoritmului de conversie** se definesc:
 - Procedurile **Expresie**, **Termen** și **Factor** conform diagramelor din fig. 5.3.1.2.a.
 - Variabile locale **op** și **op1** utilizate pentru memorarea operatorilor **aditivi** respectiv **multiplicativi**.
 - Procedura **CitesteCar** care furnizează caracterul următor din textul de analizat eliminând eventualele blankuri.
 - Variabila **ch** care în fiecare moment memorează caracterul furnizat de procedura **CitesteCar**.
 - Se consideră că expresia în format **infix** ce urmează a fi convertită este depusă ca șir de caractere în tabloul **infix** parcurs cu ajutorul indicelui **i**.
 - Forma **postfix** a expresiei se assemblează în tabloul **post** utilizând în acest scop indicele **j**.
 - Indicii **i** și **j** trebuiesc inițializați în programul principal care apelează procedura **Expresie**.
 - Pentru funcționarea corectă a procedurii de conversie, **expresia de procesat** trebuie precizată între **paranteze**.

5.3.2. Algoritmi de divizare

5.3.2.1. Tehnica divizării ("Divide and Conquer")

- Una dintre **metodele fundamentale de proiectare a algoritmilor** se bazează pe **tehnica divizării** ("divide and conquer").
- **Principiul de bază** al acestei tehnici este următorul:
 - (1) Se **descompune** (divide) problema de rezolvat în mai multe subprobleme a căror rezolvare este mai simplă și din soluțiile cărora se poate **asambla** simplu soluția problemei inițiale.
 - (2) Se repetă **recursiv** pasul (1) până când subproblemele devin banale iar soluțiile lor evidente.

- O aplicație **tipică** a **tehnicii divizării** se bazează pe **modelul recursiv** prezentat în [5.3.2.1.a].

{Tehnica divizării - soluția recursivă - Pascal}

```
PROCEDURE Rezolva(x);
BEGIN
  IF *x este divizibil în subprobleme THEN
    BEGIN
      *divide pe x în mai multe părți:
       $x_1, x_2, \dots, x_k$ ;
      Rezolva( $x_1$ ); Rezolva( $x_2$ ), ...; Rezolva( $x_k$ );
      *combină cele k soluții parțiale într-o
      soluție pentru x
    END{IF}
  ELSE
    *rezolvă pe x direct
  END; {Rezolva}
```

[5.3.2.1.a]

/* Tehnica divizării - soluția recursivă - C */

```
void rezolva(x);
{
  if (*x este divizibil în subprobleme)
  {
    *divide pe x în mai multe părți:  $x_1, x_2, \dots, x_k$ ;
    rezolva( $x_1$ );
    rezolva( $x_2$ );
    ...
    rezolva( $x_k$ );
    *combină cele k soluții parțiale într-o soluție
    pentru x
  } /*if*/
  else
    *rezolvă pe x direct;
}; /*rezolva*/
/*-----*/
```

- Dacă **recombinarea** soluțiilor parțiale este substanțial mai simplă decât rezolvarea întregii probleme, această tehnică conduce la proiectarea unor algoritmi într-adevăr eficienți.
- Datorită celor k apeluri recursive, **arborele de apeluri** asociat procedurii Rezolvă este de ordinul k.

5.3.2.2. Analiza algoritmilor de divizare

- Se presupune că timpul de execuție al rezolvării problemei de dimensiune n este $T(n)$.

- În condițiile în care prin divizări succesive problema de rezolvat devine suficient de redusă ca dimensiune, se poate considera că pentru $n \leq c$ (c constant), determinarea soluției necesită un **timp de execuție constant**, adică $\Theta(1)$.
- Se notează cu $D(n)$ timpul necesar **divizării** problemei în subprobleme și cu $C(n)$ timpul necesar **combinării** soluțiilor parțiale.
- Dacă problema inițială se divide în k **subprobleme**, fiecare dintre ele de dimensiune $1/b$ din dimensiunea problemei originale, se obține următoarea formulă recurentă [5.3.2.2.a].

$$T(n) = \begin{cases} \Theta(1) & \text{dacă } n \leq c \\ k \cdot T(n/b) + D(n) + C(n) & \text{pentru } n > c \end{cases} \quad [5.3.2.2.a]$$

5.3.2.3. Exemplu de algoritm de divizare

- Un prim exemplu de **algoritm de divizare** îl constituie **metoda de sortare Quicksort** deja prezentată (§.3.2.6).
- În acest caz problema de rezolvat se divide de fiecare dată în **două subprobleme**, rezultând un **arbore binar de apeluri**.
- **Combinarea soluțiilor** parțiale **nu** este necesară deoarece scopul este atins prin modificările care se realizează chiar de către rezolvările parțiale.

5.3.2.4. Algoritm pentru determinarea extremelor valorilor componentelor unui tablou linier. Soluția iterativă și soluția recursivă

- Algoritmul pentru determinarea **extremelor valorilor componentelor unui tablou linier** este proiectat o și procedură **Domeniu**($a, i, j, mic, mare$) care atribuie parametrilor mic și $mare$ elementul **minim** respectiv **maxim** al tabloului a din domeniul delimitat de indicii i și j respectiv ($a[i]..a[j]$).
- O **implementare iterativă** evidentă a algoritmului apare în secvența [5.3.2.4.a] sub denumirea de **DomeniuIt**.

{Algoritm pentru determinarea extremelor unui vector - soluția iterativă - varianta Pascal}

```
PROCEDURE DomeniuIt(a: TipTablou; i,j: integer;
                   VAR mic,mare: integer);
VAR k:integer;
BEGIN
    mic:= a[i]; mare:= a[i];
```

```

FOR k:=i+1 TO j DO                                     [5.3.2.4.a]
    BEGIN
        IF a[k]>mare THEN mare:= a[k];
        IF a[k]<mic THEN mic:= a[k]
    END
END; {DomeniuIt}
-----
/* Algoritm pentru deteterminarea extremelor unui vector -
soluția iterativă - varianta C */

#define n 100      /*dimensiunea tabloului*/
typedef int tiptablou[n-1];

void domeniu_it(tiptablou const a, int i,int j, int* mic,
int* mare)
{
    int k;
    *mic=a[i]; *mare=a[i];
    for( k=i+1; k <=j; k++)          /*[5.3.2.4.a]*/
    {
        if (a[k]>*mare) *mare=a[k];
        if (a[k]<*mic) *mic=a[k];
    }
} /*domeniu_it*/
/*-----*/

```

- Procedura baleează întregul vector comparând fiecare element cu cel mai mare respectiv cel mai mic element până la momentul curent.
- Este ușor de văzut că **costul** procedurii **DomeniuIt** în termenii numărului de comparații dintre elementele tabloului este $2 \cdot n - 2$ pentru un vector cu n elemente.
- Este de asemenea de observat faptul că fiecare element al lui a se va compara de **două** ori, odată pentru aflarea maximului, iar a doua oară pentru aflarea minimului.
- Acest algoritm **nu** ține cont de faptul că orice element luat în considerare drept candidat pentru minim (care apare în succesiunea de valori a lui mic) nu poate fi niciodată candidat pentru maxim, exceptând condiția de inițializare și reciproc.
 - Astfel algoritmul risipește un efort considerabil examinând fiecare element de două ori. Această risipă se poate evita.
- **O soluția recursivă** bazată pe **tehnica divizării** ar putea fi următoarea [AH85]:
 - Se împarte tabloul în două părți.
 - Se determină prin apelul recursiv al procedurii, valorile **minime** și **maxime** ale fiecăruia dintre cele două tablouri.
 - Se compară valorile **minime** și **maxime** determinate pentru subtablouri, stabilindu-se minimul și maximul absolut.

- În continuare se procedează în aceeași manieră reducând de fiecare dată la jumătate dimensiunea subtablourilor.
- Pentru dimensiuni 1 sau 2 ale subtablourilor soluția este imediată.
- O ilustrare a acestei tehnici apare în procedura recursivă **Domeniu** [5.3.2.4.b].

{Algoritm pentru deteterminarea extremelor unui vector -
soluția recursivă bazată pe tehnica divizării - Pascal}

```

CONST n = ...;
TYPE TipTablou = ARRAY[1..n] OF integer;
VAR a: TipTablou;

PROCEDURE Domeniu(a: TipTablou; i,j: integer;
                  VAR mic,mare: integer);
VAR mijloc,mic1,mare1,mic2,mare2: integer;
BEGIN
  IF j<=i+1 THEN {tablou de dimensiune 1 sau 2}
    BEGIN
      IF a[i]<a[j] THEN
        BEGIN
          mic:= a[i]; mare:= a[j]
        END
      ELSE
        BEGIN [5.3.2.4.b]
          mic:= a[j]; mare:= a[i]
        END
      END{IF}
    ELSE
      BEGIN
        mijloc:= (i+j) DIV 2;
        Domeniu(a,i,mijloc,mic1,mare1);
        Domeniu(a,mijloc+1,j,mic2,mare2);
        IF mare1>mare2 THEN mare:= mare1 ELSE
          mare:= mare2;
        IF mic1<mic2 THEN mic:= mic1 ELSE mic:= mic2
        END{ELSE}
      END; {Domeniu}
    END;
  END;

```

/* Algoritm pentru deteterminarea extremelor unui vector -
soluția recursivă bazată pe tehnica divizării- varianta C */

```

#define n 100
typedef int tiptablou[n];
tiptablou a;

void domeniu(tiptablou const a, int i,int j,int* mic,int*
mare)
{
  int mijloc,mic1,mare1,mic2,mare2;
  if (j<=i+1) /*tablou de dimensiune 1 sau 2*/
  {
    if (a[i-1]<a[j-1])

```



```

        {
            *mic=a[i-1]; *mare=a[j-1];
        }
        else
        {
            /*[5.3.2.4.b]*/
            *mic=a[j-1]; *mare=a[i-1];
        }
    } /*if*/
else
{
    mijloc=(i+j) / 2;
    domeniu(a,i,mijloc,&mic1,&mare1);
    domeniu(a,mijloc+1,j,&mic2,&mare2);
    if (mare1>mare2)
        *mare=mare1;
    else
        *mare=mare2;
    if (mic1<mic2)
        *mic=mic1;
    else
        *mic=mic2;
} /*else*/
} /*domeniu*/
/*-----*/

```

- Se constată **analogia** evidentă cu structura de principiu prezentată în secvența [5.3.2.1.a].
- În figura 5.3.2.4.a. apare **reprezentarea grafică** a modului de lucru al procedurii Domeniu.

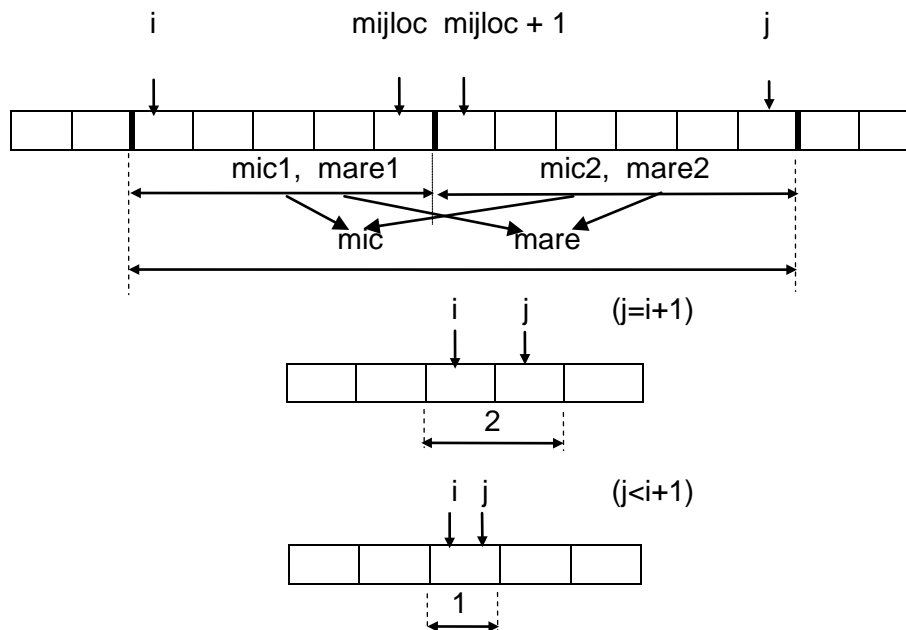


Fig.5.3.2.4.a. Funcționarea de principiu a procedurii Domeniu.

- **Urma execuției** procedurii pentru $n=10$ apare în figura 5.3.2.4.b.

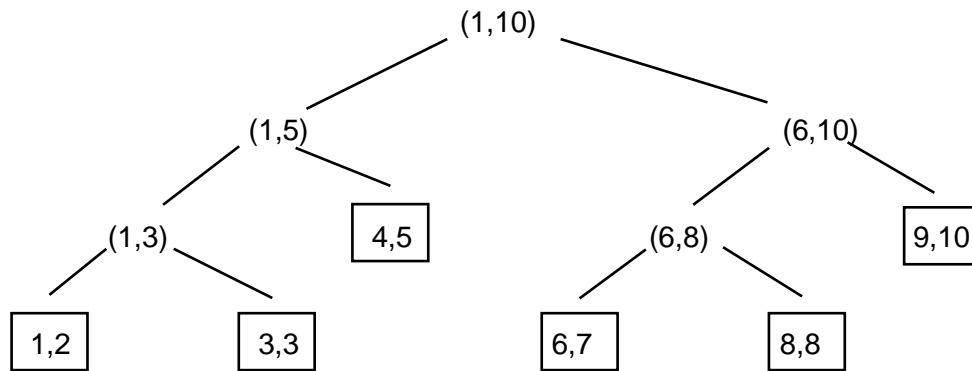


Fig.5.3.2.4.b. Arbore de apeluri al procedurii **Domeniu** pentru $n=10$

- În figură apar precizate limitele domeniilor pentru fiecare dintre apelurile procedurii.
- În chenar sunt precizate domeniile de lungime 1 sau 2 care presupun comparații directe.
- După cum se observă, arborele de apeluri este un arbore binar deoarece se execută două apeluri recursive din corpul procedurii.
- Dacă se realizează o analiză mai aprofundată a modului de implementare se constată că:
 - **Contextele** apelurilor se **salvează** în **stiva sistem** în ordinea rezultată de parcurgerea în **preordine** a arborelui de apeluri.
 - **Contextele** se **extrag** din stivă conform parcurgerii în **postordine** a arborelui de apeluri.
 - **Restaurarea** contextului fiecărui apel face posibilă parcurgerea ordonată a **tuturor posibilităților** evidențiate de arborele de apeluri.
- Per ansamblu **regia** unui astfel de algoritm recursiv este mai ridicată decât a celui iterativ (sunt necesare 11 apeluri ale procedurii, pentru $n=10$),
- Totuși, autorii demonstrează că din punctul de vedere al costului calculat relativ la **numărul de comparații** ale elementelor tabloului, el este mai eficient decât algoritmul iterativ.
 - Pornind de la observațiile:
 - (1) Procedura include comparații directe numai pentru subtablourile scurte (de lungime 1 sau 2).

- (2) Pentru subtablourile mai lungi se procedează la divizarea intervalului comparând numai extremele acestora.
- În [AH85] se indică o valoare aproximativă pentru cost (număr de comparații directe) egală cu $(3/2)n - 2$ unde n este dimensiunea tabloului analizat.
- Analiza cantitativă a algoritmului **Domeniu** se poate realiza prin particularizarea relației [5.3.2.2.a].
 - Se observă imediat că $k=2$, $b=2$, iar $D(n)$ și $C(n)$ sunt ambele $\Theta(1)$.
- În consecință rezultă ecuația [5.3.2.4.c] a cărei soluție apare în [5.3.2.4.d].

$$T(n) = \begin{cases} \Theta(1) & \text{dacă } n \leq 2 \\ 2 \cdot T(n/2) + 2 & \text{dacă } n > 2 \end{cases} \quad [5.3.2.4.c]$$

$$\Theta(T(n)) = O(n) \quad [5.3.2.4.d]$$

5.3.3. Algoritmi de reducere

- O altă categorie de algoritmi care se pretează pentru o **abordare recursivă** o reprezintă **algoritmii de reducere**.
 - Acești algoritmi se bazează pe **reducerea** în manieră **recursivă** a **gradului de dificultate** al problemei, pas cu pas, până în momentul în care aceasta devine banală.
 - În continuare se **revine** în aceeași manieră **recursivă** și se **asamblează** soluția integrală.
- În această categorie pot fi încadrați algoritmul pentru **calculul factorialului** și algoritmul pentru **rezolvarea problemei Turnurilor din Hanoi**.
- Se atrage atenția că spre deosebire de **algoritmii cu revenire** (&5.4), în acest caz **nu** se pune problema **revenirii** în caz de nereușită, element care încadrează acest tip de algoritmi într-o categorie separată.
- În continuare se prezintă un **exemplu** de algoritm de reducere.

5.3.3.1. Exemplu de algoritm de reducere. Determinarea permutărilor a n numere date

- Dându-se o secvență de n numere, se cere să se proiecteze un algoritm pentru determinarea **tuturor permutărilor celor n numere**.
- Pentru determinarea **tuturor permutărilor** se poate utiliza **tehnica reducerii**.
- **Principiul** este următorul:
 - Pentru a obține permutările de n elemente este suficient să se **fixeze** pe rând câte un element și să se permute toate celelalte $n-1$ elemente.
 - Procedând **recursiv** în această manieră se ajunge la permutări de 1 element care sunt banale.
- Această tehnică este implementată de procedura **Permut(k)** care realizează permutarea a k numere.
- Schița de principiu a acestui algoritm apare în secvența [5.3.3.1.a].

 {Schița de principiu a algoritmului pentru determinarea permutărilor a n numere naturale}

```
Procedura Permuta(k:integer);
  dacă k=1 atunci
    *afișează tabloul {s-a finalizat o permutare}
  altfel
    pentru i=1 la k execută [5.3.3.1.a]
      *interschimba pe a[i] cu a[k];
      Permuta(k-1); {apel recursiv}
      *interschimba pe a[i] cu a[k] {refacere stare}
    □
  □
```

- Numerele se presupun memorate în tabloul $a[i]$ ($i=1, 2, \dots, n$).
- Dacă $k \neq 1$, atunci fiecare dintre elementele tabloului situate pe poziții **inferioare** lui k sunt aduse (fixate) pe poziția k prin **interschimbarea** pozițiilor i și k ale tabloului a în cadrul unei bucle repetitive pentru $i=1, 2, \dots, k$.
- Inițial k ia valoarea n pentru permutarea a n numere.
- Pentru fiecare schimbare (fixare) se **apelează** recursiv rutina cu parametrul $k-1$.
 - După revenire, se **reface** starea inițială reluând în sens invers interschimbarea pozițiilor i și k .
 - Acest lucru este necesar, deoarece fixarea elementului următor presupune **refacerea** stării inițiale în vederea parcurgerii în mod ordonat, pentru fiecare element în parte a **tuturor** posibilităților.

- În momentul în care $k = 1$, se consideră **terminat** un apel și se afișează tabloul a care conține o permutare.
 - Aceasta este **condiția** care limitează adâncimea apelurilor recursive determinând **revenirea** în apelurile anterioare.
- În secvențele următoare apar două variante de implementare una Pascal și o a doua C a algoritmului de determinare a permutărilor a n numere date [5.3.3.1.b].

 {Exemplu de implementare a algoritmului pentru determinarea permutărilor a n numere naturale - varianta Pascal}

```
PROCEDURE Permuta(k: integer);
  VAR i,x: integer;
  BEGIN
    IF k=1 THEN afișează ELSE
      BEGIN
        FOR i:=1 TO k DO
          BEGIN
            x:= a[i]; a[i]:= a[k]; a[k]:= x;
            Permuta(k-1);
            x:= a[i]; a[i]:= a[k]; a[k]:= x
          END
        END
      END
    END; {Permuta}
```

 /* Exemplu de implementare a algoritmului pentru determinarea permutărilor a n numere naturale - varianta C */

```
void permuta(int k)
{
  int i,x;
  if (k==1)
    afiseaza;
  else
  {
    for (i=1; i <=k; i ++ )
    {
      x=a[i]; a[i]=a[k]; a[k]=x;
      permuta(k-1);
      x=a[i]; a[i]=a[k]; a[k]=x;
    }
  }
} /*permuta*/
/*-----*/
```

- În figura 5.3.3.1.a este reprezentat **arborele de apeluri** al procedurii **Permuta** (4) pentru permutările obținute prin fixarea primului element.
 - Pentru celelalte elemente, din lipsă de spațiu, sunt doar sugerați subarborii corespunzători.

- Structura **arborelui apelurilor** este mai **complicată** datorită faptului că **apelurile** recursive se realizează dintr-o buclă **FOR** cu o limită (k) care **descrește** odată cu creșterea nivelului arborelui.
- **Înălțimea** arborelui de apeluri este egală cu n.

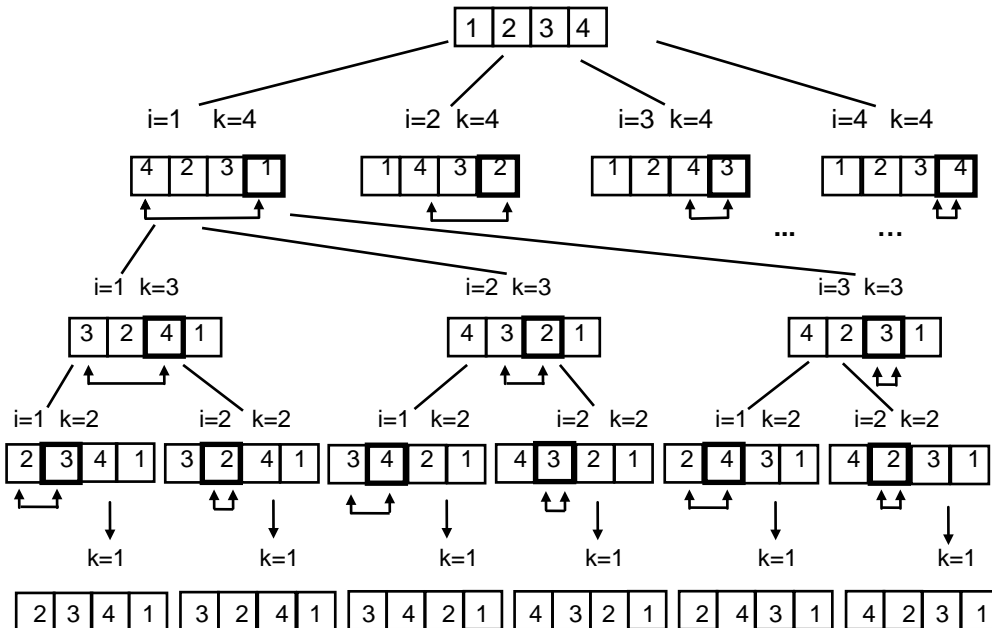


Fig.5.3.3.1.a. Arborele de apeluri pentru **Permuta** (4)

5.3.4. Algoritmi recursivi pentru determinarea tuturor soluțiilor unor probleme

- Algoritmii recursivi au **proprietatea** de a putea evidenția în mod ordonat **toate posibilitățile** referitoare la o situație dată.
- Se prezintă în acest sens două exemple de astfel de algoritmi:
 - Primul exemplu reliefează proprietatea de a evidenția în mod ordonat **toate posibilitățile** referitoare la o situație dată.
 - Cel de-al doilea exemplu **selectează** din mulțimea tuturor posibilităților evidențiate, **pe cele care prezintă interes**.

5.3.4.1. Algoritm pentru evidențierea tuturor posibilităților de tăiere a unui fir de lungime dată

- Algoritmul ce urmează evidențiază toate posibilitățile de secționare a unui fir de lungime întreagă dată (n) în părți de lungime 1 sau 2.
- Acest algoritm se bazează pe următoarea **tehnică** de lucru:
 - Pentru lungimea $n > 1$ există două posibilități:
 - Se **taie** o parte de **lungime 1** și restul lungimii ($n-1$) se secționează în **toate modurile posibile**.
 - Se **taie** o parte de **lungime 2** și restul lungimii ($n-2$) se secționează în **toate modurile posibile**.
 - Pentru lungimea $n = 1$ avem cazul banal al unei tăieturi de lungime 1.
 - Pentru lungimea $n = 0$ nu există nici o posibilitate de tăietură.
- Schița de principiu a acestui algoritm apare în secvența [5.3.4.1.a].

{Schița de principiu a algoritmului de tăiere a firului}

```

Procedura Taie(lungimeFir);
  dacă lungimeFir > 1 atunci
    *se taie o bucată de lungime 1;
    Taie(lungimeFir-1);
    *se taie o bucată de lungime 2;
    Taie(lungimeFir-2);
    *se anulează tăietura
  altfel                                     [5.3.4.1.a]
    dacă lungimeFir = 1 atunci
      *se taie bucata de lungime 1;
    □
    *afișare
  □

```

- În secvența [5.3.4.1.b] se prezintă o variantă de implementare Pascal iar în secvența următoare o variantă C.
 - În ambele implementări:
 - Procedura **Taie** implementează tehnica de lucru anterior prezentată cu precizarea că la atingerea dimensiunii $n = 1$ sau $n = 0$ se consideră procedura terminată și se afișează secvența de tăieturi generată.
 - Tăieturile sunt reprezentate grafic utilizând caracterul '.' pentru segmentul de lungime 1 și caracterul '_' pentru segmentul de lungime 2.
 - Pentru memorarea tăieturilor se utilizează un tablou de caractere z , parcurs cu ajutorul indicelui k , în care se depun caracterele corespunzătoare tăieturilor.
-

{Variantă Pascal de implementare a algoritmului de tăiere a firului}

```
VAR i,k,x: integer;
    z: ARRAY[1..9] OF char;

PROCEDURE Taie(lungimeFir: integer);
BEGIN
    IF lungimeFir>1 THEN
        BEGIN
            k:= k+1;
            z[k]:= '.';
            Taie(lungimeFir-1);
            z[k]:= '_';
            Taie(lungimeFir-2);
            k:= k-1{anulare tăietură}
        END{IF}
    ELSE
        BEGIN
            Write(' ');
            FOR i:= 1 TO k DO Write(z[i]);
            IF lungimeFir=1 THEN Write('.');
            Writeln
        END{ELSE}
    END;{Taie}
```

[5.3.4.1.b]

/* Variantă C de implementare a algoritmului de tăiere a firului */

```
int i,k,x;
char z[9];
void taie(int lungime_fir)
{
    if (lungime_fir>1)
    {
        k=k+1;
        z[k-1]='.';
        taie(lungime_fir-1);
        z[k-1]='_';
        taie(lungime_fir-2);
        k=k-1;          /*anulare tăietură*/
    } /*if*/
    else
    {
        printf(" ");
        for (i=1; i <=k; i ++){
            printf("%c", z[i-1]);
            if (lungime_fir==1)
                printf(".");
            printf("\n");
        } /*else*/
    } /*taie*/
}

/*-----*/
```


- În figura 5.3.4.1.a apare rezultatul apelului procedurii **Taie** pentru $n = 4$ respectiv $n = 5$.
 - Din această figură se poate deduce ușor maniera de execuție a procedurii (urma execuției) luând în considerare faptul că în reprezentarea grafică:
 - (1) Caracterul '.' reprezintă apelul procedurii **Taie** pentru $n-1$ (tăietură de lungime 1).
 - (2) Caracterul '-' reprezintă apelul procedurii **Taie** pentru $n-2$ (tăietură de lungime 2).

lungimeFir = 4

```

. . . .
. . -
. - .
- . .
- -

```

lungimeFir = 5

```

. . . . .
. . . -
. . - .
. - . .
. - -
- . . .
- . -
- - .

```

Fig.5.3.4.1.a. Execuția procedurii Taie pentru $n = 4$ și $n = 5$

- În figura 5.3.4.1.b se prezintă **arborele de apeluri** al procedurii Taie pentru $n = 4$.

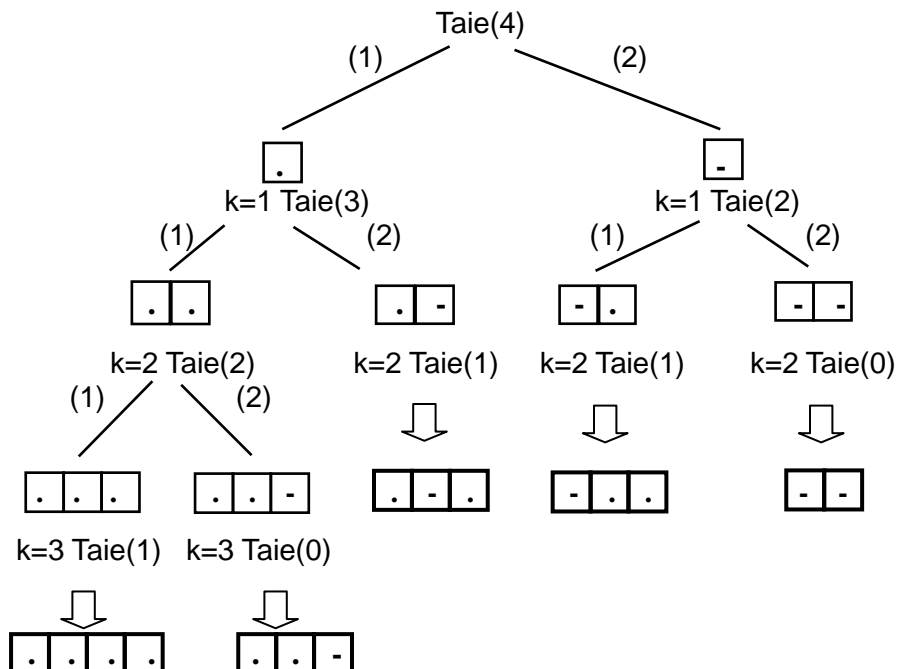


Fig.5.3.4.1.b. Arborele de apeluri al procedurii Taie (4)

- Se observă că în fiecare nod al acestui arbore sunt precizate:
 - Succesiunea bucaților tăiați.
 - Valoarea curentă a lui k .
 - Apelul recursiv care se realizează.
 - În chenar îngroșat sunt încadrate secvențele care se afișează.
- În baza modelului prezentat, acest algoritm poate fi extins cu ușurință pentru a determina posibilitățile de tăiere pentru **orice număr precizat de tăieturi**, pentru **orice structură de lungimi**.

5.3.4.2. Algoritm pentru determinarea tuturor drumurilor de ieșire dintr-un labirint

- Algoritmul care va fi prezentat în continuare presupune un **labirint**.
 - Labirintul este descris cu ajutorul unui **tablou** bidimensional de caractere de dimensiuni $(n+1) * (n+1)$.
 - Cu ajutorul caracterului ' * ' sunt reprezentați **pereții labirintului**.
 - Cu ajutorul caracterului ' ' (spațiu liber) sunt reprezentate **culoarele**.
 - Punctul de start este **centrul labirintului**.
- **Drumul de ieșire** se caută cu ajutorul unei **proceduri recursive Cauta** (x, y) unde x și y sunt **coordonatele** locului curent în labirint și în același timp **indici** ai tabloului care materializează labirintul.
- **Căutarea** se execută astfel:
 - (1) Dacă valoarea locului curent este ' ' (spațiu liber):
 - Se intră pe un **posibil** traseu de ieșire.
 - Se marchează locul cu caracterul ' # '.
 - Dacă s-a ajuns la margine rezultă că s-a găsit un **drum de ieșire** din labirint și se execută afișarea tabloului bidimensional (labirintul și drumul găsit).
 - (2) Dacă valoarea locului curent **nu** este ' ' (spațiu liber):

- Se apelează recursiv procedura **Cauta** pentru cele patru puncte din vecinătatea imediată a locului curent, după direcțiile axelor de coordonate (dreapta, sus, stânga, jos).
- Pentru fiecare căutare reușită traseul apare marcat cu ' # '.
- Marcarea asigură **nu** numai memorarea drumului de ieșire dar în același timp înlătură **reparcurgerea** unui drum deja ales.
 - Reluarea parcurgerii aceluiași drum poate conduce la un ciclu **infinit**.
- Este importantă sublinierea faptului că marcajul de drum se **șterge** de îndată ce s-a ajuns într-o **fundătură** sau dacă s-a găsit o **ieșire**, în ambele situații **revenindu-se** pe drumul parcurs până la proximal punct care permite selecția unei **noi posibilități** de drum.
- **Ștergerea** se execută prin generarea unui caracter ' ' pe poziția curentă înainte de părăsirea procedurii.
 - Aceasta corespunde practic înfășurării "firului parcurgerii" conform metodei "firului Ariadnei".
- În secvența [5.3.4.2.a] este prezentată schița de principiu a procedurii **Caută** iar în [5.3.4.2.b] o variantă de implementare Pascal și una C.

{Schița de principiu a algoritmului de căutare a drumului de ieșire dintr-un labirint}

```
Procedura Cauta(x,y: coordonate);
  dacă *locul este liber atunci                                [5.3.4.2.a]
    *marcheaza locul
    dacă *s-a ajuns la ieșire atunci *afișează drumul
    altfel
      Cauta(x+1,y); {dreapta} Cauta(x,y+1); {sus}
      Cauta(x-1,y); {stânga} Cauta(x,y-1) {jos}
    □
    *șterge marcajul
  □
```

{Variantă Pascal de implementare a algoritmului de căutare a drumului de ieșire dintr-un labirint}

```
PROCEDURE Cauta(x,y: integer);
  BEGIN
    IF m[x,y]=' ' THEN
      BEGIN
        m[x,y]:= '#';
        IF ((x MOD n)=0)OR((y MOD n)=0) THEN *afișează
        ELSE
          BEGIN
            Cauta(x+1,y); Cauta(x,y+1); Cauta(x-1,y);
            Cauta(x,y-1)
          END;
        END;
      END;
```

```

        m[x,y]:= ' ' {ștergere marcaj}
    END
END;{Cauta}
-----
/* Variantă C de implementare a algoritmului de căutare a
drumului de ieșire dintr-un labirint */

void cauta(int x,int y)
{
    if (m[x][y]==' ')
    {
        /*[5.3.4.2.b]*/
        m[x][y]='#';
        if (((x % n)==0) || ((y % n)==0)) ;
            *afiseaza*
        else
        {
            cauta(x+1,y);
            cauta(x,y+1);
            cauta(x-1,y);
            cauta(x,y-1);
        }
        m[x][y]=' '; /* ștergere marcaj */
    }
    /*cauta*/
}
/*-----*/

```

- Procedura recursivă **Cauta** materializează metoda expusă anterior.
- Dacă în timpul căutării se atinge una din extremele valorilor abscisei sau ordonatei (0 sau n), s-a găsit o ieșire din labirint și se realizează afișarea.
- În figura 5.3.4.2.a se prezintă un exemplu de execuție al acestei proceduri.

```

*****
*      *
*  **  *
* *   *
* *   *
* *   *
* *   *
* *   *
* *   *
* *   *
*****

*****
*####*
*####*
*####*
*####*
*####*
*####*
*####*
*####*
*####*
*****

*****
*      *
*  **  *
* *   *
* *   *
* *   *
* *   *
* *   *
* *   *
* *   *
*****

```

Fig.5.3.4.2.a. Exemplu de execuție al procedurii Cauta

5.3.5. Algoritmi pentru trasarea unor curbe recursive

- **Curbele recursive** reprezintă un alt domeniu de aplicație al algoritmilor recursivi.
 - Astfel de curbe pot fi generate elegant cu ajutorul unui sistem de calcul, întrucât ele presupun un număr redus de module care se îmbină succesiv, conform unor reguli simple, bine precizate.
- Scopul acestui paragraf este acela de a prezenta doi algoritmi recursivi care permit generarea unor astfel de curbe.
- Se face precizarea că studiul curbelor recursive care aparent **nu** are o aplicabilitate practică imediată, contribuie în mod esențial la fundamentarea, consolidarea și înțelegerea conceptului de recursivitate.
- Se face de asemenea precizarea că aceste curbe se pot include și în categoria **fractalilor**.

5.3.5.1. Algoritm recursiv pentru generarea curbelor lui Hilbert

- În figura 5.3.5.1.a apar reprezentate patru curbe notate H_0 , H_1 , H_2 și H_3 care reprezintă curbele lui Hilbert de ordin 0, 1, 2 și 3.
- Se observă că:
 - O curbă Hilbert H_{i+1} se obține prin compunerea a patru instanțe de curbe H_i .
 - Cele 4 instanțe au dimensiunea înjumătățită.
 - Ele sunt rotite într-o manieră specifică.
 - Instanțele sunt legate între ele prin trei linii de legătură (prezentate îngroșat în desen).

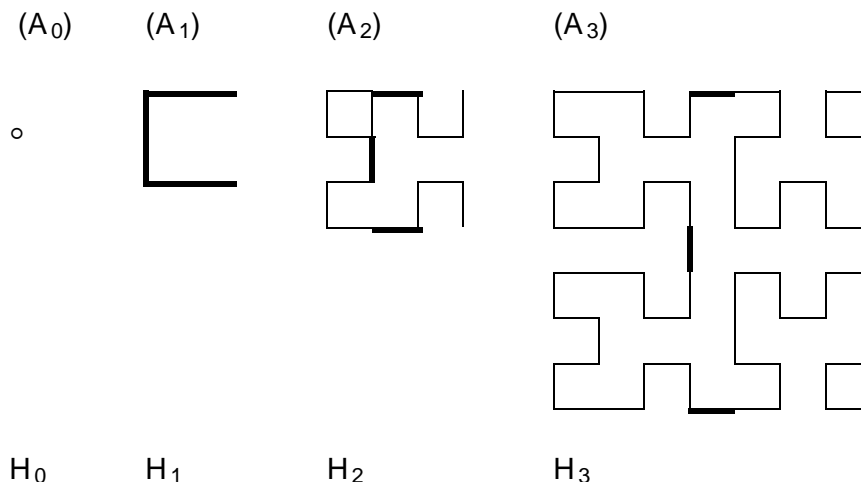


Fig.5.3.5.1.a. Curbele lui Hilbert de ordin 0, 1, 2 și 3

- Se mai observă că H_1 poate fi considerată ca și constând din 4 instanțe al curbei vide H_0 , unite prin aceleași linii de legătură.
- Acestea sunt **curbele Hilbert** datând din 1891.
- Se presupune că există disponibilă o **bibliotecă grafică** care dispune de operatorii:
 - **MoveTo**(x, y) care poziționează cursorul grafic în punctul de coordonate x, y ;
 - **LineTo**(x, y) care trasează o linie dreaptă între poziția curentă și cea indicată prin coordonatele parametri x, y .
- În continuare ne propunem să concepem un program recursiv simplu pentru **generarea și reprezentarea grafică a curbelor Hilbert** de diferite ordine.
 - Deoarece fiecare curbă H_i constă din patru copii înjumătățite H_{i-1} , este normal ca procedura care-l desenează pe H_i să fie compusă din patru părți, fiecare desenând câte o curbă H_{i-1} rotită corespunzător și la dimensiunea cerută.
- Prin generalizare se deduc cele patru **modele recursive** care sunt prezentate în figura 5.3.5.1.b.
- Cele patru părți sunt denumite A, B, C și D, iar punctele lor de start sunt marcate cu cerușe.
- Rutinele care trasează liniile sunt reprezentate prin săgeți indicând direcția de trasare.

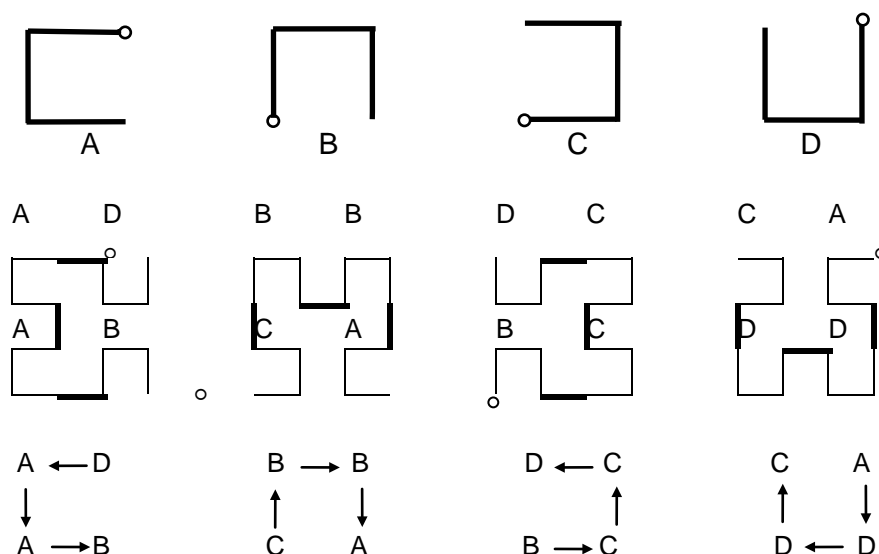


Fig.5.3.5.1.b. Modele recursive pentru curbele lui Hilbert

- Pornind de la aceste **modele recursive** se deduc imediat **schemele recursive generice** care permit generarea curbelor lui Hilbert [5.3.5.1.a].

```
-----
A: D ← A ↓ A → B
B: C ↑ B → B ↓ A
C: B → C ↑ C ← D
D: A ↓ D ← D ↑ C
-----
```

[5.3.5.1.a]

- Dacă **lungimea** unității de linie se notează cu h , procedura care corespunde modelului A poate fi redactată utilizând apeluri recursive la ea însăși precum și la procedurile D și B realizate în aceeași manieră [5.3.5.1.b].

```
-----
{Procedura pentru trasarea recursivă a modelului A al unei
curbe Hilbert - varianta Pascal}
```

```
PROCEDURE A(i: integer);
BEGIN
  IF i>0 THEN
    BEGIN
      D(i-1); x:= x-h; LineTo(x,y);
      A(i-1); y:= y-h; LineTo(x,y);
      A(i-1); x:= x+h; LineTo(x,y);
      B(i-1)
    END
  END; {A}
```

[5.3.5.1.b]

- Procedura este inițializată în programul principal pentru fiecare curbă Hilbert care va fi suprainprimată.
 - Se consideră că pe același desen se suprapun curbe Hilbert de mai multe ordine succesive.
 - Ceea ce rezultă se aseamănă cu reprezentările **fractalilor**.
- Programul principal determină **punctul inițial de start** și **lungimea curentă a liniei** h pentru fiecare curbă generată.
- Variabila h_0 care precizează dimensiunea totală a paginii trebuie să aibă valoarea 2^k unde $k > n$, n fiind ordinul maxim al curbelor care suprainprimă.
- Programul integral care trasează curbele Hilbert H_1, H_2, \dots, H_n apare în secvența [5.3.5.1.c] în varaintă Pascal respectiv C.

```
-----
{Program pentru trasarea recursivă a unor curbe Hilbert de
diferite ordine -varianta Pascal}
```

```
PROGRAM Hilbert;
      {desenează curbele hilbert de ordin 1,2,...,n}
```

```

CONST n=4; h0=512;
VAR i,h,x,y,x0,y0: integer;
PROCEDURE B(i: integer); FORWARD;
PROCEDURE C(i: integer); FORWARD;
PROCEDURE D(i: integer); FORWARD;

PROCEDURE A(i: integer); {desenează modelul A}
BEGIN
  IF i>0 THEN
    D(i-1); x:= x-h; LineTo(x,y);
    A(i-1); y:= y-h; LineTo(x,y);
    A(i-1); x:= x+h; LineTo(x,y);
    B(i-1)
  END
END; {A} [5.3.5.1.c]

PROCEDURE B(i: integer); {desenează modelul B}
BEGIN
  IF i>0 THEN
    BEGIN
      C(i-1); y:= y+h; LineTo(x,y);
      B(i-1); x:= x+h; LineTo(x,y);
      B(i-1); y:= y-h; LineTo(x,y);
      A(i-1)
    END
  END
END; {B}

PROCEDURE C(i: integer); {desenează modelul C}
BEGIN
  IF i>0 THEN
    BEGIN
      B(i-1); x:= x+h; LineTo(x,y);
      C(i-1); y:= y+h; LineTo(x,y);
      C(i-1); x:= x-h; LineTo(x,y);
      D(i-1)
    END
  END
END; {C}

PROCEDURE D(i: integer); {desenează modelul D}
BEGIN
  IF i>0 THEN
    BEGIN
      A(i-1); y:= y-h; LineTo(x,y);
      D(i-1); x:= x-h; LineTo(x,y);
      D(i-1); y:= y+h; LineTo(x,y);
      C(i-1)
    END
  END
END; {D}

BEGIN {programul principal}
  i:= 0; h:= h0; x0:= h DIV 2; y0:= x0;
  REPEAT {trasează curba hilbert de ordin i}
    i:= i+1; h:= h DIV 2;
    x0:= x0+(h DIV 2); y0:= y0+(h DIV 2);
    x:= x0; y:= y0; MoveTo(x,y);

```



```

        A(i)
    UNTIL i=n;
END.
-----
/* Program pentru trasarea recursivă a unor curbe Hilbert de
diferite ordine -varianta C */

/*deseneaza curbele hilbert de ordin 1,2,...,n*/
#include <graphics.h>

#define n 4
#define h0 512

int i,h,x,y,x0,y01;

void b(int);
void c(int);
void d(int);
void graphic_init(void);
void graphic_close(void);

void a(int i) /*desenează modelul A*/
{
    if (i>0)
    {
        d(i-1); x=x-h; lineto(x,y);
        a(i-1); y=y-h; lineto(x,y);
        a(i-1); x=x+h; lineto(x,y);
        b(i-1);
    }
} /*a*/ /*[5.3.5.1.c]*/

void b(int i) /*desenează modelul B*/
{
    if (i>0)
    {
        c(i-1); y=y+h; lineto(x,y);
        b(i-1); x=x+h; lineto(x,y);
        b(i-1); y=y-h; lineto(x,y);
        a(i-1);
    }
} /*b*/

void c(int i) /*desenează modelul C*/
{
    if (i>0)
    {
        b(i-1); x=x+h; lineto(x,y);
        c(i-1); y=y+h; lineto(x,y);
        c(i-1); x=x-h; lineto(x,y);
        d(i-1);
    }
} /*c*/

void d(int i) /*desenează modelul D*/

```

```

    {
        if (i>0)
        {
            a(i-1); y=y-h; lineto(x,y);
            d(i-1); x=x-h; lineto(x,y);
            d(i-1); y=y+h; lineto(x,y);
            c(i-1);
        }
    } /*d*/

int main(int argc, const char* argv[])
{
    /*programul principal*/
    graphic_init();
    i=0;
    h=h0;
    x0=h/2;
    y01=x0;
    do { /*trasează curba hilbert de ordin i*/
        i++;
        h=h / 2;
        x0=x0+(h/2);
        y01=y01+(h/2);
        x=x0; y=y01;
        moveto(x,y);
        a(i);
    } while (!(i==n));
    getchar();
    graphic_close();
    return 0;
}

/*OS specific code falows*/
/*BorlandC 3.1 exemple*/
void graphic_init()
{
    /* request auto detection */
    int gdriver =DETECT, gmode, errorcode;
    /* initialize graphics mode */
    initgraph(&gdriver, &gmode, "");
    /* read result of initialization */
    errorcode =graphresult();
    if (errorcode !=grOk) /* an error occurred */
    {
        printf("Graphics error: %s\n",
               grapherrormsg(errorcode));
        exit(1);
    }
}

void graphic_close()
{
    closegraph();
}

/*-----*/

```

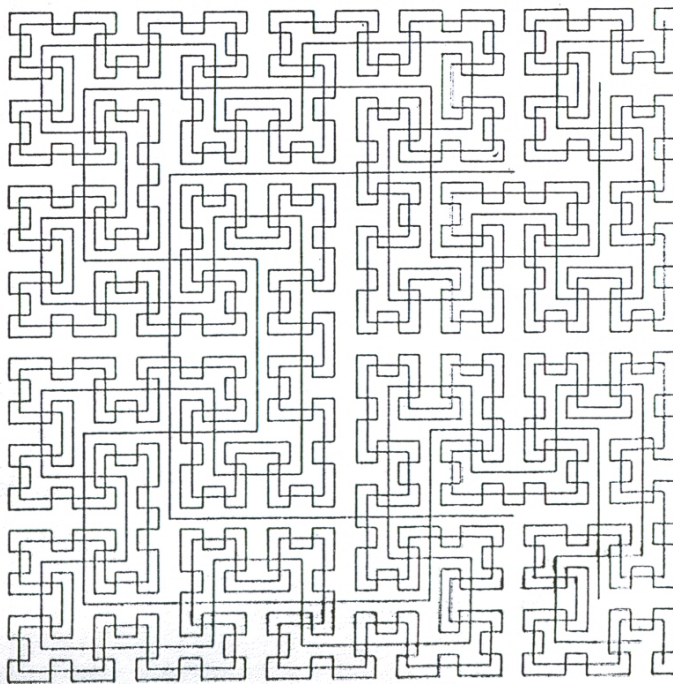


Fig.5.3.5.1.c. Curbele lui Hilbert $H_1 - H_5$

- În figura 5.3.5.1.c sunt reprezentate curbele lui Hilbert până la ordinul 5, rezultate din execuția programului `Hilbert` prezentat drept exemplu .

5.3.5.2. Algoritm recursiv pentru generarea curbelor lui Sierpinski

- Un exemplu mai complicat îl reprezintă **curbele lui Sierpinski** care apar în figura 5.3.5.2.a (S_1 și S_2).

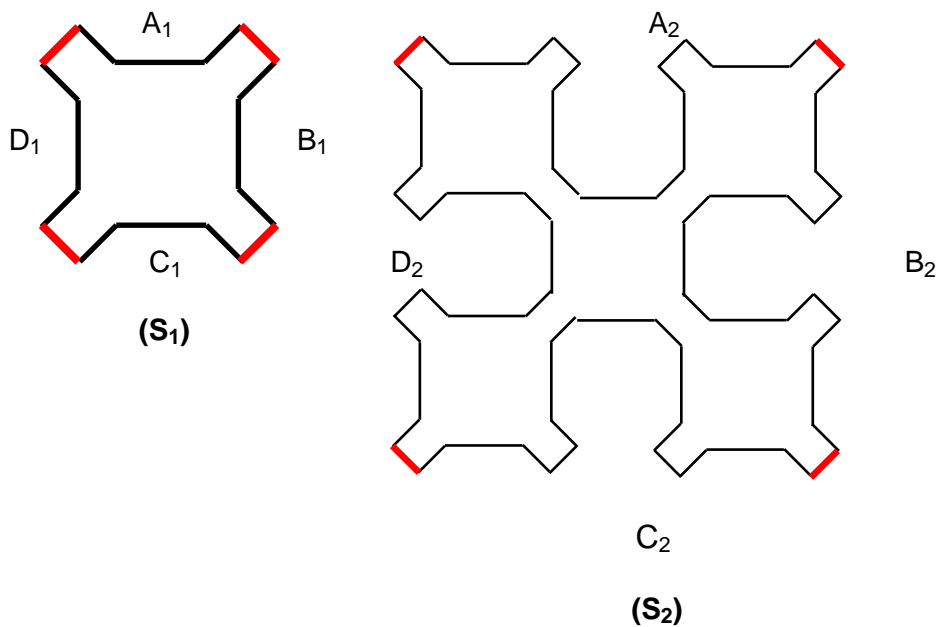


Fig.5.3.5.2.a. Modele recursive pentru curbele lui Sierpinski de ordin 1 și 2

- Diferența dintre cele două categorii de curbe este următoarea: curbele lui Sierpinski sunt curbe **închise**, în timp ce curbele lui Hilbert sunt curbe **deschise**.
- În consecință, în cazul curbelor lui **Sierpinski**:
 - **Modelul recursiv** de bază trebuie să fie o **curbă deschisă**.
 - Cele patru părți ale sale sunt conectate prin **linii de închidere** care **nu** aparțin **modelului recursiv** propriu-zis.
 - **Liniile de închidere** constau în patru linii drepte situate în colțurile figurii și care sunt reprezentate îngroșat în desen (fig.5.3.5.2.a).
 - Din reprezentare rezultă clar **modul de asamblare** al curbelor de ordin inferior pentru a obține o curbă de ordin superior (fig.5.3.5.2.a).
- Pornind de la aceste observații, pot fi construite cele patru **modele generice** ale **curbelor lui Sierpinski**.
- Ele sunt denumite A, B, C și D iar liniile interne de legătură sunt precizate explicit prin săgeți (fig.5.3.5.2.b).
- Cele patru **modele generice recursive** sunt **identice** exceptând faptul că sunt **rotite** cu 90° fiecare.
 - După cum se observă **modelele generice** sunt **curbe deschise**.
 - Este foarte important de subliniat faptul că cele patru modele se assemblează prin intermediul unui **model de bază**, format din **linii de închidere** care este reprezentat în aceeași figură deasupra (fig.5.3.5.2.b).

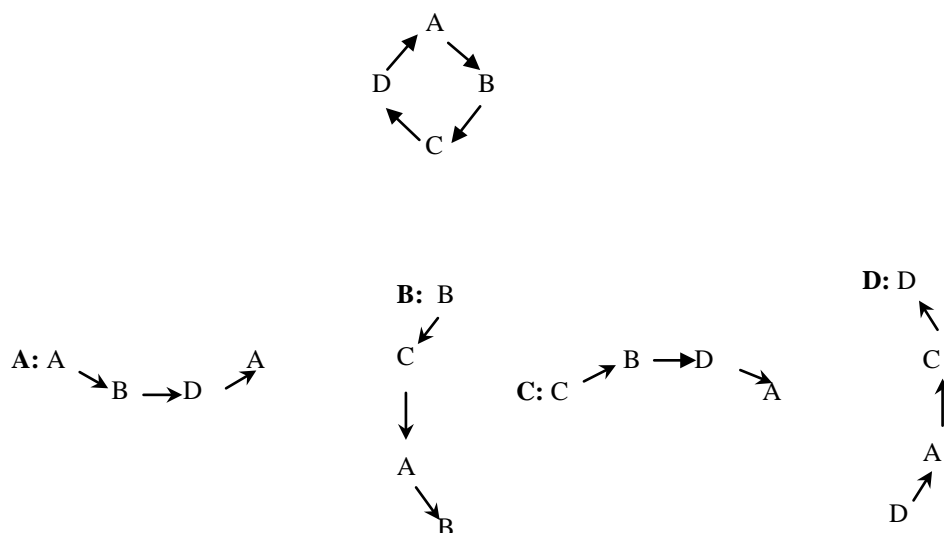


Fig.5.3.5.2.b. Modelele recursive ale curbelor lui Sierpinski

- Din aceste **modele recursive** pot fi deduse simplu **schemele recursive** care permit generarea curbelor.
- În secvența [5.3.5.2.a] apare **schema modelului de bază**, iar în [5.3.5.2.b] **schemele recursive** ale curbelor componente, unde o săgeată dublă indică o linie de lungime dublă.

S: $A \rightarrow B \leftarrow C \leftarrow D \rightarrow$ [5 . 3 . 5 . 2 . a]

$A : A \rightarrow B \Rightarrow D \rightarrow A$
 $B : B \leftarrow C \Downarrow A \rightarrow B$ [5 . 3 . 5 . 2 . b]
 $C : C \leftarrow D \Leftarrow B \leftarrow C$
 $D : D \rightarrow A \Uparrow C \leftarrow D$

- Considerând aceleași facilități de reprezentare ca și în cazul anterior pot fi redactate cu ușurință rutinele recursive care trasează curbele lui Sierpinski.
- Astfel, procedura **A** poate fi construită pornind de la prima linie a modelului recursiv precizat în [5.3.5.2.b] la fel și procedurile **B**, **C** și **D**.
- În secvența [5.3.5.2.c] apare un model de implementare pentru procedura **A**.

{Exemplu de implementare a modelului A pentru o curba Sierpinski varianta Pascal}

```

PROCEDURE A(i: integer);
BEGIN
  IF i>0 THEN
    BEGIN
      A(i-1); x:= x+h; y:= y-h; LineTo(x,y);
      B(i-1); x:= x+2*h; LineTo(x,y);
      D(i-1); x:= x+h; y:= y+h; LineTo(x,y);
      A(i-1)
    END
  END
END; {A}

```

- **Programul principal** este conceput conform modelului [5.3.5.2.a].
- Sarcina **programului principal** este aceea de a iniția generarea succesivă de curbe Sierpinski de ordine crescătoare și de a fixa pentru fiecare, valoarea lui h și valorile coordonatelor inițiale conform cu dimensiunea zonei de afișare.
- Tot în sarcina programului principal cade și trasarea celor patru linii care închid curbele trasate (modelul de bază).
- Programul integral apare în [5.3.5.2.d].

- Eleganța și oportunitatea utilizării recursivității în această situație este evidentă.
- Corectitudinea programului poate fi ușor dedusă din structura sa și din forma modelelor.
- Limitarea adâncimii recursivității s-a realizat cu ajutorul parametrului *i*, garantând în acest mod terminarea programului.
- Utilizarea iterației în aceste situații conduce la programe complicate și greoaie.

{Program pentru trasarea curbelor Sierpinski de diferite ordine - varianta Pascal}

```

PROGRAM Sierpinski;
  {trasează curbele Sierpinski de ordin 1,2,...,n}

  CONST n=4; h0=512;

  VAR i,h,x,y,xo,yo: integer;

  PROCEDURE B(i: integer); FORWARD;           [5.3.5.2.d]
  PROCEDURE C(i: integer); FORWARD;
  PROCEDURE D(i: integer); FORWARD;

  PROCEDURE A(i: integer); {desenează modelul A}
  BEGIN
    IF i>0 THEN
      BEGIN
        A(i-1); x:= x+h; y:= y-h; LineTo(x,y);
        B(i-1); x:= x+2*h; LineTo(x,y);
        D(i-1); x:= x+h; y:= y+h; LineTo(x,y);
        A (i-1)
      END
    END; {A}

  PROCEDURE B; {desenează modelul B}
  BEGIN
    IF i>0 THEN
      BEGIN
        B(i-1); x:= x-h; y:= y-h; LineTo(x,y);
        C(i-1); y:= y-2*h; LineTo(x,y);
        A(i-1); x:= x+h; y:= y-h; LineTo(x,y);
        B(i-1)
      END
    END; {B}

  PROCEDURE C;           {desenează modelul C}           [5.3.5.2.d]
  BEGIN
    { . . . }
  END; {C}

  PROCEDURE D; {desenează modelul D}
  BEGIN
    { . . . }
  
```

```

    END; {D}

BEGIN {programul principal}
    i:= 0; h:= ho DIV 4; xo:= 2*h; yo:= 3*h;
    REPEAT
        i:= i+1; xo:= xo-h;
        h:= h DIV 2; yo:= yo+h;
        x:= xo; y:= yo; MoveTo(x,y);
        A(i); x:= x+h; y:= y-h; LineTo(x,y); {traseaza liniile
        B(i); x:= x-h; y:= y-h; LineTo(x,y); de legatura}
        C(i); x:= x-h; y:= y+h; LineTo(x,y);
        D(i); x:= x+h; y:= y+h; LineTo(x,y)
    UNTIL i=n;
END.

```

```

-----
/* Program pentru trasarea curbelor Sierpinski de diferite
ordine - varianta C */

```

```

#include <graphics.h>

```

```

    /*trasează curbele Sierpinski de ordin 1,2,...,n*/
enum { n =4, h0 =512};
int i,h,x,y,xo,yo;

```

```

void b(int i);
void c(int i);
void d(int i);
void graphic_init(void);
void graphic_close(void);

```

```

void a(int i) /*desenează modelul A*/
{
    if (i>0){
        a(i-1); x=x+h; y=y-h; lineto(x,y);
        b(i-1); x=x+2*h; lineto(x,y);
        d(i-1); x=x+h; y=y+h; lineto(x,y);
        a(i-1);
    }
    /*a*/
}

```

```

void b(int i) /*desenează modelul B*/
{
    if (i>0){
        b(i-1); x=x-h; y=y-h; lineto(x,y);
        c(i-1); y=y-2*h; lineto(x,y);
        a(i-1); x=x+h; y=y-h; lineto(x,y);
        b(i-1);
    }
    /*b*/
}

```

```

void c(int i) /*desenează modelul C*/ /*[5.3.5.2.d]*/
{
    /*. . .*/
}
/*c*/

```

```

void d(int i) /*desenează modelul D*/
{
    /*. . .*/
} /*d*/

int main(int argc, const char* argv[])
{
    /*programul principal*/
    graphic_init();
    i=0;
    h=h0/4;
    xo=2*h;
    yo=3*h;
    do {
        i++; xo=xo-h;
        h=h/2; yo=yo+h;
        x=xo; y=yo; moveto(x,y);
        a(i); x=x+h; y=y-h; lineto(x,y); /*traseaza liniile*/
        b(i); x=x-h; y=y-h; lineto(x,y); /*de legatura*/
        c(i); x=x-h; y=y+h; lineto(x,y);
        d(i); x=x+h; y=y+h; lineto(x,y);
    } while (!(i==n));
    graphic_close();
    return 0;
}

/*OS specific code falows*/
/*BorlandC 3.1 exemple*/
void graphic_init()
{
    /* request auto detection */
    int gdriver =DETECT, gmode, errorcode;
    /* initialize graphics mode */
    initgraph(&gdriver, &gmode, "");
    /* read result of initialization */
    errorcode =graphresult();
    if (errorcode !=grOk) /* an error occurred */
    {
        printf("Graphics error: %s\n",
               grapherrormsg(errorcode));
        exit(1);
    }
}

void graphic_close()
{
    closegraph();
}

/*-----*/

```

- În figura 5.3.5.2.c apare un exemplu graphic de execuție a programului de generare și reprezentare a curbelor lui Sierpinski pentru $n = 4$.

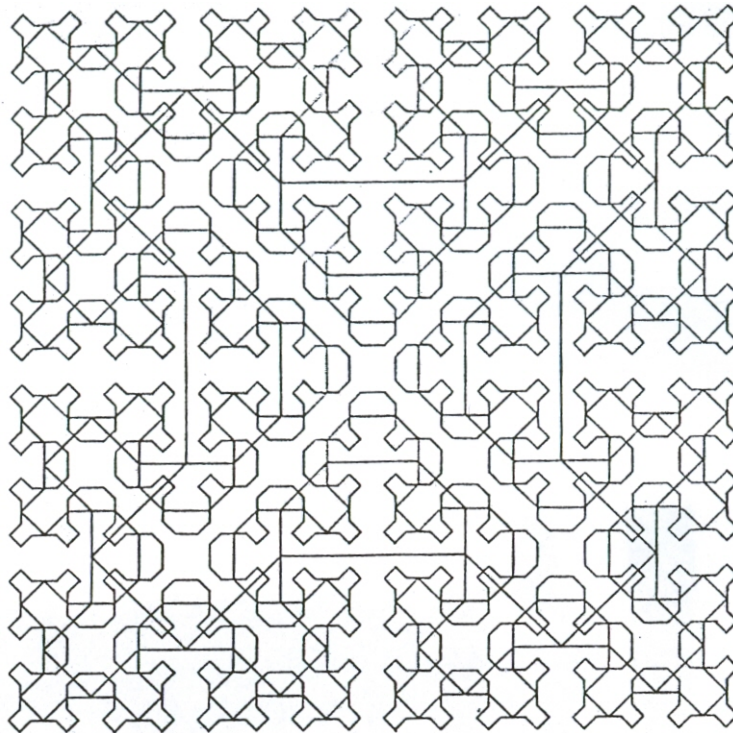


Fig.5.3.5.2.c. Curbele lui Sierpinski de ordinele $S_1 - S_4$

5.4. Algoritmi cu revenire (backtracking)

- Unul din subiectele de mare interes ale programării se referă la **rezolvarea** unor probleme cu **caracter general**.
 - Ideea este de a concepe **algoritmi generali** pentru găsirea soluțiilor unor probleme specifice, care să **nu** se bazeze pe un **set fix de reguli de calcul**, ci pe **încercări repetate** și **reveniri** în caz de nereușită.
- Modalitatea comună de realizare a acestei tehnici constă în **descompunerea obiectivului** (taskului) în **obiective parțiale** (taskuri parțiale).
 - De regulă această descompunere este exprimată în mod natural în **termeni recursivi** și constă în explorarea unui **număr finit** de subtaskuri.
 - În general, întregul proces poate fi privit ca un proces de **încercare** sau **căutare** care **construiește** în mod gradat **soluția** și **parcurge** în același timp un **arbore de subprobleme**.

- Obținerea unor **soluții parțiale** sau **finale** care **nu** satisfac, provoacă **revenirea recursivă** în cadrul procesului de căutare și **reluarea** acestuia până la obținerea **soluției dorite**.
- Din acest motiv, astfel de algoritmi se numesc **algoritmi cu revenire** ("**backtracking algorithms**").
- În multe cazuri arborii de căutare cresc foarte rapid, de obicei **exponențial**, iar efortul de căutare crește în aceeași măsură.
- În mod obișnuit, arborii de căutare pot fi simplificați numai cu ajutorul **euristicilor**, simplificări care se reflectă de fapt în restrângerea volumului de calcul și încadrarea sa în **limite acceptabile**.
- **Scopul** acestui paragraf:
 - **Nu** este acela de a discuta principiile și regulile generale ale **euristicii**.
 - Mai degrabă:
 - (1) Se vor aborda **principiile partajării unei probleme în subproblemele care o rezolvă**.
 - (2) Se va utiliza **recursivitatea** drept suport al soluționării acestora.

5.4.1. Turneul calului

- **Specificarea problemei turneului calului:**
 - Se consideră o tablă de șah (eșicher) cu n^2 câmpuri.
 - Un cal căruia îi este permis a se mișca conform regulilor șahului, este plasat în câmpul cu coordonatele inițiale x_0 și y_0 .
 - Se cere să se găsească acel **parcurs al calului** - dacă există vreunul - care acoperă toate câmpurile tablei, **trecând o singură dată** prin fiecare.
 - **Primul pas** în abordarea problemei parcurgerii celor n^2 câmpuri este acela de a considera problema următoare:
 - "La un moment dat se încearcă execuția unei **mișcări următoare** sau se constată că **nu** mai este posibilă nici una și atunci se **revine** în pasul anterior".
 - Pentru început se va defini algoritmul care încearcă să execute **mișcarea următoare** a calului.
-

{Schița algoritmului pentru realizarea mișcării următoare a calului - varianta pseudocod Pascal}

```
PROCEDURE ÎncearcăMișcareaUrmătoare;
BEGIN
    *inițializează lista mișcărilor
    REPEAT
        *selectează posibilitatea următoare din lista
        mișcărilor
        IF *este acceptabilă THEN
            BEGIN
                [5.4.1.a]
                *înregistrează mișcarea curentă;
                IF *tabela nu e plină THEN
                    BEGIN
                        ÎncearcăMișcareaUrmătoare;
                        IF NOT *mișcare reușită THEN
                            *șterge înregistrarea curentă
                        END
                    ELSE
                        *mișcare reușită
                    END
                UNTIL (*mișcare reușită) OR (*nu mai există
                posibilități în lista mișcărilor)
            END; {ÎncearcăMișcareaUrmătoare}
```

 /* Schița algoritmului pentru realizarea mișcării următoare
 a calului - varianta pseudocod C */

```
void încearcă_mișcarea_următoare()
{
    /*inițializează lista mișcărilor */
    do {
        *selectează posibilitatea următoare din lista
        mișcărilor
        if (este acceptabilă)
            {
                /*[5.4.1.a]*/
                *înregistrează mișcarea curentă;
                if (tabela nu e plină)
                    {
                        încearcă_mișcarea_următoare();
                        if (!mișcare reușită)
                            *șterge înregistrarea curentă*/;
                    }
                else;
                *mișcare reușită
            }
    } while (!(mișcare reușită) || (nu mai există
    posibilități în lista mișcărilor));
} /*încearcă_mișcarea_următoare*/
/*-----*/
```

- În continuare se vor preciza **structurile de date** care vor fi utilizate.
- În primul rând **tabla de șah** va fi reprezentată prin matricea *t* pentru care se introduce tipul *TipIndice* [5.4.1.b].

{Definirea structurilor de date: tabla de șah}

```
TYPE TipIndice = 1..n; [5.4.1.b]
    TipTabela = ARRAY[TipIndice,TipIndice] OF integer;
```

```
enum {n =10};
typedef unsigned tipindice; /*[5.4.1.b]*/
typedef int tiptabela[n][n];
/*-----*/
```

- După cum se observă, în câmpurile tabeli *t* se memorează valori întregi și **nu** valori booleene care să indice ocuparea.
- Acest lucru se face cu scopul de a păstra **urma traseului parcurs** conform următoarei **convenții** [5.4.1.c].

{Convenția de marcarea a traseului parcurs de cal pe tabela de șah}

```
t[x,y] = 0; {câmpul (x,y) nu a fost încă vizitat}
t[x,y] = i; {câmpul (x,y) a fost vizitat în pasul i
            (1 ≤ i ≤ n2)} [5.4.1.c]
```

- În continuare se stabilesc **parametrii de apel** ai procedurii. Aceștia trebuie să precizeze:
 - (1) **Condițiile de start** ale noii mișcări (parametri de intrare).
 - (2) Dacă mișcarea respectivă este **reușită** sau **nu** (parametru de ieșire).
- Pentru prima cerință (1) sunt necesare:
 - **Coordonatele x, y** de la care va porni mișcarea.
 - **Valoarea i** care precizează **numărul mutării curente** (din rațiuni de înregistrare și evidență).
- Pentru cea de-a doua cerință (2) se introduce **parametrul de ieșire** *q* = **true** desemnând mișcare reușită respectiv *q* = **false** mișcare nereușită, din punctul de vedere al acceptării mișcării.
- Problema care se pune în continuare este aceea a **rafinării** propozițiilor precedate de caracterul "*" în secvența pseudocod [5.4.1.a].
 - În primul rând faptul că **tabela* nu este plină se exprimă prin $i < n^2$.
 - Dacă se introduc **variabilele locale** *u* și *v* pentru a preciza coordonatele unei **posibile destinații a mișcării** conform regulilor după care se efectuează saltul calului, atunci:

- Propoziția *este acceptabilă poate fi exprimată ca și o combinație logică a condițiilor $1 \leq u \leq n$ și $1 \leq v \leq n$ (adică noul câmp este pe tabelă) și că el **nu** a fost vizitat anterior ($t[u,v] = 0$).
- Aserțiunea *înregistrează mișcarea devine $t[u,v] = i$.
- Aserțiunea *șterge înregistrarea curentă, se exprimă prin $t[u,v] = 0$.
- Se mai introduce **variabila booleană locală** q1 utilizată ca și parametru rezultat intermediar pentru apelul recursiv al procedurii. Variabila q1 **substituie** de fapt condiția *mișcare reușită.
- Se ajunge în definitiv la următoarea formulare a procedurii [5.4.1.d].

{Rafinarea procedurii Încearcă - pasul 1 de rafinare - varianta Pascal}

```

PROCEDURE Incearca(i: integer; x,y: indice;
                  VAR q: boolean);
VAR u,v: integer; q1: boolean;
BEGIN
  *inițializează lista mișcărilor
  REPEAT
    *fie u,v coordonatele mișcării următoare conform
    regulilor șahului
    IF (1<=u<=n) AND (1<=v<=n) AND (t[u,v]=0) THEN
      BEGIN
        t[u,v]:= i;                                [5.4.1.d]
        IF i<n*n THEN
          BEGIN
            Incearca(i+1,u,v,q1);
            IF NOT q1 THEN t[u,v]:= 0
          END
        ELSE
          q1:= true
        END
      UNTIL q1 OR (nu mai există posibilități în lista
                  mișcărilor);

      q:= q1
    END; {Incearcă}

```

/* Rafinarea procedurii Încearcă - pasul 1 de rafinare - varianta C */

```

typedef tipindice indice;
typedef unsigned int boolean;
#define true (1)
#define false (0)

void incearca(int i, indice x, indice y, boolean* q)
{
  int u,v; boolean q1;

```

```

*initializează lista mișcărilor
do {
    *fie u,v coordonatele mișcării următoare conform
    regulilor șahului
    if ((1<=u<=n)&&(1<=v<=n)&&(t[u][v]==0))
    {
        t[u][v]=i;                                /*[5.4.1.d]*/
        if (i<n*n)
        {
            incearca(i+1,u,v,&q1);
            if (!q1) t[u][v]=0;
        }
        else
            q1=true;
    }
} while (!(q1||(*nu mai există posibilități în lista
mișcărilor)));

*q=q1;
}/*incearca*/
/*-----*/

```

- Relativ la această secvență se fac următoarele precizări.
- **Tehnica** utilizată este cea cunoscută în literatura de specialitate sub denumirea de **tehnică "look-ahead" (tehnica scrutării)**. În baza acestei tehnici:
 - (1) Se apelează procedura cu **coordonatele curente** x și y .
 - (2) Se selectează o **nouă mișcare de coordonate** u și v (de fapt următoarea mișcare din cele 8 posibile din lista de mișcări).
 - (3) Se încearcă **realizarea mișcării următoare** plecând de la poziția u, v .
 - Dacă mișcarea **nu** este reușită, respectiv s-au parcurs fără succes toate cele 8 posibilități ale listei de mișcări plecând de la u și v , se **anulează mișcarea curentă** (u, v), ea fiind lipsită de perspectivă (bucla **REPEAT**).
- Privind din perspectiva evoluției căutării, fiecare dintre cele 8 mișcări este tratată în manieră similară:
 - Pornind de la fiecare dintre mișcări se merge atât de departe cât se poate.
 - În caz de nereușită se încercă mișcarea următoare din lista de mișcări până la epuizarea tuturor posibilităților.
 - Dacă s-au epuizat toate posibilitățile, se **anulează** mișcarea curentă.
- După cum se observă, procedura se extinde de fapt peste **trei niveluri de căutare**, element care îi permite **revenirea** în caz de eșec în vederea selectării unui nou parcurs.

- **Arborele de apeluri** asociat căutării are următoarele caracteristici:
 - Este de **ordinul 8** (din fiecare punct se pot selecta 8 posibilități de mișcare).
 - Are **înălțimea** n^2 (numărul de pași necesari pentru soluția finală), element care explică **complexitatea** procesului de determinare a soluției problemei.
- În pasul următor și ultimul de rafinare mai rămân câteva puncte de specificat.
- Precizarea **saltului calului**.
 - Fiind dată o poziție inițială $\langle x, y \rangle$ există opt **posibilități** pentru generarea coordonatelor destinației mișcării următoare $\langle u, v \rangle$, care sunt numerotate de la 1 la 8 în fig.5.4.1.a.

	3		2	
4				1
		X		
5				8
	6		7	

Fig.5.4.1.a. Mișcările posibile ale calului pe eșicher

- O metodă simplă de obținere a lui u și v din x și y este de a aduna la acestea din urmă, pentru fiecare dintre cele 8 posibilități de salt, a unor **diferențe specifice de coordonate** memorate în două tablouri, unul pentru coordonata x notat cu a și unul pentru coordonata y notat cu b .
 - Indicele k precizează **numărul următorului candidat**, respectiv **numărul mișcării următoare** din lista mișcărilor ($1 \leq k \leq 8$)
- O formă a celor două tabele a și b apare în [5.4.1.e].

```

a[1]:= 2; b[1]:= 1;
a[2]:= 1; b[2]:= 2;
a[3]:=-1; b[3]:= 2;
a[4]:=-2; b[4]:= 1;
a[5]:=-2; b[5]:=-1;
a[6]:=-1; b[6]:=-2;
a[7]:= 1; b[7]:=-2;
a[8]:= 2; b[8]:=-1;

```

[5 . 4 . 1 . e]

- Detaliile de implementare apar în programul [5.4.1.e], variantă Pascal respectiv C.
- Procedura recursivă este **inițiată** printr-un apel cu coordonatele x_0, y_0 de la care pornește parcursul turneului calului.

- Acestui câmp i se atribuie valoarea 1, restul câmpurilor se marchează ca fiind libere (valoare nulă).
- Se face următoarea precizare: o variabilă $t[u,v]$ există numai dacă u și v sunt în domeniul $1..n$.
 - În varianta Pascal a programului această cerință a fost implementată cu ajutorul operatorului **IN**, utilizând mulțimea S , implementare care pentru valori mici ale lui u respectiv v este foarte eficientă[5.4.1.f].
 - În varianta C, această cerință a fost implementată prin comparații simple directe[5.4.1.f].

{Determinarea Turneului Calului - varianta finală Pascal}

```

PROGRAM TurneulCalului;
  CONST n=5;
  TYPE TipIndice = 1..n;
  VAR i,j: TipIndice;
      q: boolean;
      S: SET OF integer;
      a,b: ARRAY[1..8] OF integer;
      t: ARRAY[TipIndice,TipIndice] OF integer;

  PROCEDURE Incearca(i: integer; x,y: TipIndice;
                    VAR q: boolean);
    VAR k,u,v: integer;
        k: integer;
        q1: boolean;
  BEGIN
    k:= 0;
    REPEAT
      k:= k+1; q1:= false;
      u:= x+a[k]; v:= y+b[k];
      IF (u IN S) AND (v IN S) THEN
        IF t[u,v]=0 THEN
          BEGIN
            t[u,v]:= i;
            IF i<n*n THEN
              BEGIN
                Incearca(i+1,u,v,q1);
                IF NOT q1 THEN t[u,v]:= 0
              END
            ELSE
              q1:= true
            END
          END
        UNTIL q1 OR (k=8);
      q:= q1
    END; {Incearca}

  BEGIN {programul principal}
    S:= [1,2,3,4,5,6,7,8];
    a[1]:= 2; b[1]:= 1;
  
```



```

a[2]:= 1; b[2]:= 2;
a[3]:=-1; b[3]:= 2;
a[4]:=-2; b[4]:= 1;
a[5]:=-2; b[5]:=-1;
a[6]:=-1; b[6]:=-2;
a[7]:= 1; b[7]:=-2;
a[8]:= 2; b[8]:=-1;
FOR i:=1 TO n DO
    FOR j:= 1 TO n DO t[i,j]:= 0;
t[1,1]:= 1; Incearca(2,1,1,q);
IF q THEN
    FOR i:= 1 TO n DO
        BEGIN
            FOR j:= 1 TO n DO Write(' ',t[i,j]);
            Writeln
        END
    ELSE Writeln(' nu exista solutie ')
END.

```

/* Determinarea Turneului Calului - varianta finală C */

```

#include <limits.h>
#include <stdarg.h>
#include <stdlib.h>

```

```

enum {n =5};
typedef unsigned char tipindice;

```

```

typedef unsigned int boolean;
#define true (1)
#define false (0)

```

```

tipindice i,j;
boolean q;
int a[8],b[8];
int t[n][n];

```

```

void incearca(int i, tipindice x,tipindice y, boolean* q)
{
    int k,u,v;
    boolean q1;
    k=0;
    do {
        k=k+1;
        q1=false;
        u=x+a[k-1]; v=y+b[k-1];
        if ((0<=u<=n-1) && (0<=v<=n-1))
            if (t[u-1][v-1]==0)
            {
                t[u-1][v-1]=i;
                if (i<n*n)
                {
                    incearca(i+1,u,v,&q1);
                    if (! q1)
                        t[u-1][v-1]=0;
                }
            }
    } while (q1==0);
}

```

```

        }
        else
            q1=true;
    }
} while (!(q1 || (k==8)));
*q=q1;
} /*incearca*/

int main(int argc, const char* argv[])
{
    /*programul principal*/

    a[0]=2; b[0]=1;
    a[1]=1; b[1]=2;
    a[2]=-1; b[2]=2;
    a[3]=-2; b[3]=1;
    a[4]=-2; b[4]=-1;
    a[5]=-1; b[5]=-2;
    a[6]=1; b[6]=-2;
    a[7]=2; b[7]=-1;
    for(i=1; i<=n; i++)
        for(j=1; j<=n; j++)
            t[i-1][j-1]=0;
    t[0][0]=1;
    incearca(2,1,1,&q);
    if (q)
        for( i=1; i <=n; i++)
        {
            for( j=1; j <=n; j++)
                printf(" %3i", t[i-1][j-1]);
            printf("\n");
        }
        else printf("nu exista solutie \n");
    getch();
    return 0;
}
/*-----*/

```

- În figura 5.4.1.b se prezintă rezultatele execuției programului TurneulCalului pentru pozițiile inițiale (1,1), (3,3) cu n = 5 și (1,1) cu n = 6.

1	6	15	10	21	23	10	15	4	25
14	9	20	5	16	16	5	24	9	14
19	2	7	22	11	11	22	1	18	3
8	13	24	17	4	6	17	20	13	8
25	18	3	12	23	21	12	7	2	19

1	16	7	26	11	14
34	25	12	15	6	27
17	2	33	8	13	10
32	35	24	21	28	5
23	18	3	30	9	20
36	31	22	19	4	29

Fig.5.4.1.b. Exemple de execuție ale programului TurneulCalului

- **Caracteristica esențială** a algoritmului **turneul calului** este următoarea:
 - Înaintează spre soluția finală pas cu pas, **tatonând** și **înregistrând** drumul parcurs.
 - Aceasta este **metoda euristică** cunoscută sub denumirea de **”trial and error”**.
 - Dacă la un moment dat constată că drumul ales **nu conduce la soluția dorită** ci la o fundătură, **revine ștergând** înregistrările pașilor până la proximal punct care permite o nouă posibilitate de drum.
- Această manieră de abordare a rezolvării problemei se numește **revenire** sau **”backtraking”**, iar algoritmi care o implementează se numesc **algoritmi cu revenire (backtraking)**.

5.4.2. Probleme (n,m). Determinarea unei soluții

- **Modelul** principal general al unui **algoritm de revenire** se pretează foarte bine pentru rezolvarea claselor de problemelor pentru care:
 - (1) **Soluția finală** presupune parcurgerea a n **pași** succesivi.
 - (2) Fiecare pas poate fi selectat dintre m **posibilități**.
- O astfel de problemă o vom numi **problemă de tip (n,m)**.
- În secvența [5.4.2.a] apare **modelul principal** de rezolvare a unei astfel de probleme în forma procedurii **Încearcă** .
 - Modelul oferă o **singură soluție** a problemei.

{Model principal de rezolvare a unei probleme de tip (n,m).
Determinarea unei soluții}

Procedura Încearcă;

BEGIN

 *inițializează selecția posibilităților;

REPEAT

 *selectează posibilitatea următoare;

IF acceptabilă **THEN**

BEGIN

[5.4.2.a]

```

        *înregistrează-o ca și curentă;
    IF *soluție incompletă THEN
        BEGIN
            Incearca *pasul urmator;
            IF *nu este reușit THEN
                *șterge înregistrarea curentă
            END
        ELSE
            *pas reușit (soluție completă)
        END{IF}
    UNTIL (*pas reușit) OR (*nu mai sunt posibilități)
    END; {Incearca}
-----
/* Model principal de rezolvare a unei probleme de tip
(n,m). Determinarea unei soluții */

void incearca()
{
    *initializează selecția posibilităților;
    do
        *selectează posibilitatea următoare;
        if ( acceptabila ) /*[5.4.2.a]*/
        {
            *înregistrează-o ca și curentă;
            if ( *soluție incompletă )
            {
                incearca *pasul urmator;
                if ( *nu este reușit )
                    *șterge înregistrarea curentă
            }
            else
                *pas reușit (soluție completă)
        } /*if */
    while (!( *pas reușit ) || !( *nu mai sunt posibilități ))
    } /*incearca*/
} /*-----*/

```

- Acest **model principal** poate fi concretizat în diverse forme.
 - În continuare se prezintă două variante.
- (1) În prima **variantă**:
 - Procedura **Incearca1** are drept parametru de apel **numărul pasului curent**.
 - Explorarea posibilităților se realizează în bucla interioară **REPEAT** [5.4.2.b].

{Rezolvarea unei probleme de tip (n,m). Determinarea unei soluții - Varianta 1}

```

Procedura Incearca1(i: TipPas);
    VAR posibilitate: TipPosibilitate;
    BEGIN

```

```

posibilitate:= 0;{inițializează selecția
                    posibilităților}
REPEAT
    posibilitate:= posibilitate+1;  {selecție posibilitate
                                    următoare}

    IF *acceptabila THEN
        BEGIN
            *înregistrează-o ca și curentă;
            IF i<n THEN {soluție incompletă}
                BEGIN
                    Incearca1(i+1); {încearcă pasul următor}
                    IF *nereușit THEN
                        *șterge înregistrarea curentă
                END
            ELSE
                *soluție completă (afișare)
            END
        UNTIL *soluție completă OR (posib=m)
    END;{Incearca1}
-----
/* Rezolvarea unei probleme de tip (n,m). Determinarea unei
soluții - Varianta 1 */

void incearca1(TipPas i)
{
    TipPosibilitate posib;
    posib=0; /*inițializează selecția posibilităților*/
    do
        posib=posib+1; /*selecție posibilitate următoare*/
        if ( *acceptabila )
            {
                /*[5.4.2.b]*/
                *înregistrează-o ca și curentă;
                if ( i<n ) /*soluție incompletă*/
                    {
                        incearca1(i+1); /*încearcă pasul următor*/
                        if ( *nereușit )
                            *șterge înregistrarea curentă
                    }
                else
                    *soluție completă (afișare)
            }
        while (!*soluție completă || (posib!=m))
    }/*incearca1*/
}/*-----*/

```

- (2) În cea de-a doua **variantă**:
 - Procedura **Incearca2** are drept parametru de apel o **posibilitate de selecție**.
 - Construcția soluției se realizează apelând în manieră recursivă procedura, succesiv, pentru fiecare posibilitate în parte [5.4.2.c].
- Din punctul de vedere al **finalității** cele două variante sunt **identice**, ele diferă doar ca formă.

{Rezolvarea unei probleme de tip (n,m). Determinarea unei soluții - Varianta 2}

```
Procedura Incearca2(posibilitate: TipPosibilitate);
BEGIN
  IF *acceptabilă THEN
    BEGIN
      *înregistrează-o ca și curentă;
      IF *soluție incompletă THEN
        BEGIN
          Incearca2(Posibilitate1);
          Incearca2(Posibilitate2);
          ...
          Incearca2(Posibilitatem);
          *șterge înregistrarea curentă
        END
      ELSE
        *soluție completă (afișare)
      END
    END
  END; {Incearca2}
```

/* Rezolvarea unei probleme de tip (n,m). Determinarea unei soluții - Varianta 2 */

```
void incearca2(TipPosibilitate posib)
{
  if ( *acceptabilă )
  {
    *înregistrează-o ca și curentă;
    if ( *soluție incompletă )
    {
      incearca2(Posibilitate1);
      incearca2(Posibilitate2);
      ...
      incearca2(Posibilitatem);
      *șterge înregistrarea curentă
    }
    else
      *soluție completă (afișare)
  }
} /*incearca2*/
/*-----*/
```

- Se presupune că la fiecare pas numărul posibilităților de examinat este **fix** (m) și că procedura este apelată inițial prin **Incearca2**(1).
- În continuare în cadrul acestui paragraf vor fi prezentate câteva **aplicații** ale **algoritmilor cu revenire**, care se pretează deosebit de bine unei abordări recursive.

5.4.3. Problema celor 8 regine

- **Problema celor 8 regine** reprezintă un alt exemplu bine cunoscut de utilizare a algoritmilor cu revenire.
- Această problemă a fost investigată de **K.F. Gauss** pe la 1850, care însă **nu** a rezolvat-o complet.
 - Până în prezent **nu** a fost găsită o soluție analitică completă.
- În schimb problema celor 8 regine poate fi rezolvată prin încercări repetate sistematice, activitate care necesită o mare cantitate de muncă, răbdare, exactitate și acuratețe, atribute în care mașina de calcul excelează asupra omului chiar atunci când acesta este un geniu.
- **Specificarea problemei celor 8 regine:**
 - Pe o tablă de șah trebuie plasate 8 regine astfel încât nici una dintre ele să nu le amenințe pe celelalte.
- Se observă imediat că aceasta este o problemă de tip (n, m) :
 - Există 8 regine care trebuie plasate pe eșicher, deci soluția necesită **8 pași**.
 - Pentru fiecare din cele 8 regine există, după cum se va vedea, **8 posibilități** de a fi așezate pe tabla de șah.
- Pornind de la modelul [5.4.2.a] se obține imediat următoarea **formulare primară a algoritmului** [5.4.3.a].

{Rezolvarea Problemei celor 8 regine - schița de principiu}

```
PROCEDURE Incearca(i: regina);
BEGIN
  *inițializează selecția locului de plasare pentru
    a i-a regină
  REPEAT
    *selectează locul următor
    IF *loc sigur THEN
      BEGIN
        *plasează regina i
        IF i<8 THEN
          BEGIN
            Incearca(i+1);
            IF *încercare nereușită THEN *ia regina
          END
        ELSE
          *încercare reușită (i=8)
        END
      UNTIL *încercare reușită OR (*nu mai există locuri)
    END; {Incearca}
```

```

-----
/* Rezolvarea Problemei celor 8 regine - schița de principiu
*/

void incearca(regina i)
{
    *inițializează selecția locului de plasare pentru
      a i-a regina
    do
        *selectează locul urmator
        if ( *loc sigur )
            {
                /*[5.4.3.a]*/
                *plasează regina i
                if ( i<8 )
                    {
                        incearca(i+1);
                        if ( *încercare nereușită ) *ia regina
                    }
                else
                    *încercare reușită (i=8)
            }
        while (!( *încercare reușită ) || !( *nu mai există locuri ))

    } /*incearca*/
} /*-----*/

```

- Sunt necesare câteva **precizări**.
 - Deoarece din regulile șahului se știe că regina amenință **toate** câmpurile situate pe aceeași **coloană**, **rând** sau **diagonale** în raport cu câmpul pe care ea se află, **rezultă** că fiecare **coloană** a tablei de șah va putea conține **o singură** regină.
 - Drept urmare alegerea poziției celei de-a i-a regine poate fi restrânsă **numai la coloana i**.
- În consecință:
 - Parametrul i din cadrul algoritmului devine **indexul coloanei** pe care va fi plasată regina i.
 - Procesul de selecție se **restrânge** la una din cele 8 valori posibile ale indicelui j care precizează rândul în cadrul coloanei.
- În concluzie:
 - Avem o problemă tipică (8,8).
 - Soluționarea ei necesită 8 **pași** (așezarea celor 8 regine).
 - Fiecare regină se așează într-una din cele 8 poziții ale coloanei proprii (8 **posibilități**).

- **Arborele de apeluri recursive** este de ordinul 8 și are înălțimea 8.
- În continuare se impune alegerea modalității de **reprezentare a poziției** celor 8 regine pe tabla de șah.
 - **Soluția imediată** este aceea a reprezentării **tablei de șah** cu ajutorul unei **matrice** t de dimensiuni $[8, 8]$.
 - O astfel de reprezentare conduce însă la operații greoaie și complicate de determinare a câmpurilor disponibile.
- Pornind de la principiul că de fiecare dată trebuie utilizate reprezentările directe cele mai relevante și mai eficiente ale informației, în cazul de față **nu** se vor reprezenta pozițiile reginelor pe tabla de șah ci faptul că o regină a fost sau nu plasată pe o anumită **coloană**, pe un anumit **rând**, sau pe o anumită **diagonală**.
 - Știind că pe fiecare **coloană** este plasată o singură regină, se poate alege următoarea **reprezentare a datelor** [5.4.3.b].

{Problema celor 8 regine - definirea structurilor de date}

```
VAR x: ARRAY[1..8] OF integer;
    a: ARRAY[1..8] OF boolean;
    b: ARRAY[b1..b2] OF boolean;           [5.4.3.b]
    c: ARRAY[c1..c2] OF boolean;
```

/*Problema celor 8 regine - definirea structurilor de date*/

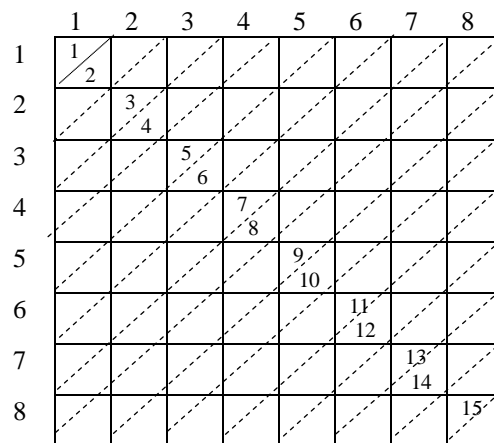
```
    int x[8];
    int y[8];                               /*[5.4.3.b]*/
    boolean b[b2];
    boolean c[c2];
/*-----*/
```

- Presupunând că regina i se plasează în poziția (i, j) pe tabla de șah, **semnificația** acestei reprezentări apare în secvența [5.4.3.c].

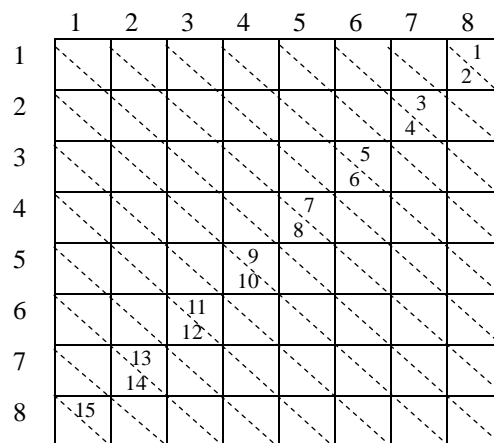
x[i]:= j precizează locul j al reginei în coloana i
a[j]:= true nici o regină nu amenință rândul j [5.4.3.c]
b[k]:= true nici o regină nu amenință diagonală /k
c[k]:= true nici o regină nu amenință diagonală \k

```
x[i]=j      precizează locul j al reginei în coloana i
a[j]=true   nici o regină nu amenință rândul j   /*[5.4.3.c]*/
b[k]=true   nici o regină nu amenință diagonală /k
c[k]=true   nici o regină nu amenință diagonală \k
/*-----*/
```

- Se precizează că pe tabela de șah există 15 diagonale / (încălate spre **dreapta**) și 15 diagonale \ (încălate spre **stânga**) (figura 5.4.3.a).
- Caracteristica unei diagonale / este aceea că **suma** coordonatelor i și j pentru oricare câmp care îi aparține este o constantă.
- Pentru diagonalele \ este caracteristic faptul că **diferența** coordonatelor i și j pentru oricare câmp al diagonalei este o constantă.
- În figura 5.4.3.a apar reprezentate aceste două tipuri de diagonale.
- După cum se observă, pentru diagonalele / **sume** $i+j$ sunt cuprinse în domeniul $[2, 16]$.
- Iar pentru diagonalele \ **diferențe** aparțin domeniului $[-7, 7]$.



$$2 \leq i + j \leq 16$$



$$-7 \leq i - j \leq 7$$

Fig.5.4.3.a. Tipuri de diagonale în problema celor 8 regine

- Aceste considerente fac posibilă alegerea valorilor limitelor b_1, b_2, c_1, c_2 pentru indicii tabelor b și c [5.4.3.b].
 - Una din posibilități este cea utilizată în continuare pentru care s-au ales $b_1 = 2, b_2 = 16, c_1 = 0, c_2 = 14$.
 - Pentru b_1 și b_2 s-au ales chiar limitele intervalului în care iau valori **sumele indicilor**.
 - Intervalul în care iau valori **diferențele indicilor** a fost translatat cu valoarea 7 spre dreapta pentru a obține valori **pozitive** pentru indicele de acces în tabela c .
- Cu alte cuvinte:
 - Accesul în tabloul b destinat evidenței diagonalelor / se realizează prin $b[i+j]$.
 - Accesul în tabloul c destinat evidenței diagonalelor \ prin $c[i-j+7]$.
 - Inițial toate locațiile tablourilor a, b și c se poziționează pe **true**.
- Cu ajutorul acestor reprezentări afirmația **plasează regina pe poziția (i, j)* , i fiind coloana proprie, devine [5.4.3.d]:

```
{*plasează regina i pe poziția (i,j)}
```

```
x[i]:= j; a[j]:= false; b[i+j]:= false;           [5.4.3.d]
c[i-j+7]:= false
```

```
/*plasează regina i pe poziția (i,j)*/
```

```
x[i]=j; a[j]=false; b[i+j]=false;           /*[5.4.3.d]*/
c[i-j+7]=false
/*-----*/
```

- În același context, afirmația **ia regina apare rafinată în secvența [5.4.3.e]:*

```
{*ia regina}
```

```
a[j]:= true; b[i+j]:= true; c[i-j+7]:= true   [5.4.3.e]
```

```
/*ia regina*/
```

```
                                           /*[5.4.3.e]*/
a[j]=true; b[i+j]=true; c[i-j+7]=true
/*-----*/
```

- Condiția **sigură* este îndeplinită dacă câmpul (i, j) destinație aparține unui rând și unor diagonale care sunt libere (**true**), situație ce poate fi exprimată de următoarea expresie logică [5.4.3.f]:
-

```
{*sigură}
```

```
    a[j] AND b[i+j] AND c[i-j+7] [5.4.3.f]
```

```
-----  
/*sigură*/
```

```
    a[j] && b[i+j] && c[i-j+7] /*[5.4.3.f]*/  
/*-----*/
```

- Programul care materializează aceste considerente apare în secvența [5.4.3.g].

```
-----  
{Problema celor 8 regine - determinarea unei soluții  
Varianta finală}
```

```
PROGRAM Reginel;
```

```
    {găsește o soluție a problemei celor 8 regine}
```

```
VAR i: integer; q: boolean;
```

```
    a: ARRAY[1..8] OF boolean;
```

```
    b: ARRAY[2..16] OF boolean;
```

```
    c: ARRAY[0..14] OF boolean;
```

```
    x: ARRAY[1..8] OF integer;
```

```
PROCEDURE Incearca(i: integer; VAR q: boolean);
```

```
    VAR j: integer;
```

```
    BEGIN
```

```
        j:= 0;
```

```
        REPEAT
```

```
            j:= j+1; q:= false;
```

```
            IF a[j] AND b[i+j] AND c[i-j+7] THEN
```

```
                BEGIN
```

```
                    x[i]:= j;
```

```
                    [5.4.3.g]
```

```
                    a[j]:= false; b[i+j]:= false;
```

```
                    c[i-j+7]:= false;
```

```
                    IF i<8 THEN
```

```
                        BEGIN
```

```
                            Incearca(i+1,q);
```

```
                            IF NOT q THEN
```

```
                                BEGIN
```

```
                                    a[j]:= true; b[i+j]:= true;
```

```
                                    c[i-j+7]:= true
```

```
                                END
```

```
                            END
```

```
                        ELSE
```

```
                            q:= true
```

```
                        END
```

```
                    UNTIL q OR (j=8)
```

```
                END; {Incearca}
```

```
BEGIN {programul principal}
```

```
    FOR i:=1 TO 8 DO a[i]:= true;
```

```
    FOR i:=2 TO 16 DO b[i]:= true;
```

```
    FOR i:=0 TO 14 DO c[i]:= true;
```

```
    Incearca(1,q);
```

```
    IF q THEN
```

```
        FOR i:=1 TO 8 DO Write(x[i]);
```

```

        Writeln
END.
-----
/*-----*/
#include <stdio.h>
/*găsește o soluție a problemei celor 8 regine*/

typedef unsigned boolean;
#define true (1)
#define false (0)

int i; boolean q;
boolean a[8];
boolean b[15];
boolean c[15];
int x[8];

void incearca(int i, boolean* q)
{
    int j;
    j=0;
    do {
        j=j+1; *q=false;
        if (a[j-1] && b[i+j-2] && c[i-j+7]){
            x[i-1]=j;                                     /*[5.4.3.g]*/
            a[j-1]=false;
            b[i+j-2]=false;
            c[i-j+7]=false;
            if (i<8){
                incearca(i+1,q);
                if (!*q){
                    a[j-1]=true;
                    b[i+j-2]=true;
                    c[i-j+7]=true;
                }
            }
            else
                *q=true;
        }
    } while (!( *q || (j==8) ));
} /*incearca*/

int main(int argc, const char* argv[])
{
    /*programul principal*/
    for( i=1; i <= 8; i++)
        a[i-1]=true;
    for( i=2; i <=16; i++)
        b[i-2]=true;
    for( i=0; i <=14; i++)
        c[i]=true;
    incearca(1,&q);
    if (q)
        for( i=1; i <=8; i++)
            printf("%i ", x[i-1]);
    printf("\n");
}

```

```

return 0;
}
/*-----*/

```

	1	2	3	4	5	6	7	8
1	R							
2							R	
3					R			
4								R
5		R						
6				R				
7						R		
8			R					

Fig.5.4.3.b. Soluția problemei celor 8 regine

- Soluția determinată de program este $x = (1, 5, 8, 6, 3, 7, 2, 4)$ și apare reprezentată grafic în figura 5.4.3.b.

5.4.4. Determinarea tuturor soluțiilor unei probleme (n,m). Generalizarea problemei celor 8 regine

- Modelul de determinare a unei soluții pentru o problemă de tip (n, m) poate fi ușor extins pentru a determina **toate soluțiile** unei astfel de probleme.
- Pentru aceasta este necesar ca **generarea pașilor** care construiesc soluția să se facă într-o manieră **ordonată** care garantează că un anumit pas **nu** poate fi generat decât o **singură** dată.
 - Această proprietate corespunde **căutării în arborele de apeluri**, într-o manieră **sistematică**, astfel încât fiecare nod să fie vizitat o singură dată.
- De îndată ce o soluție a fost găsită și înregistrată, se trece la determinarea soluției următoare pe baza unui proces de selecție sistematică până a la epuizarea **tuturor** posibilităților.
- Schema generală de principiu derivată din [5.4.2.a], care rezolvă această problemă apare în [5.4.4.a].
- În mod surprinzător, **găsirea tuturor soluțiilor unei probleme de tip (n, m)** presupune un algoritm mai **simplic** decât găsierea unei singure soluții.

{Model pentru rezolvarea unei probleme de tip (n, m) - determinarea tuturor soluțiilor}

Procedura Incearca;

BEGIN

FOR *toate posibilitățile de selecție **DO**

IF *selecție acceptabilă **THEN**

BEGIN

*înregistrează-o ca și curentă [5.4.4.a]

IF *soluția incompletă **THEN**

Incearca *pasul urmator

ELSE

*evidențiază soluția;

*șterge înregistrarea curentă

END

END; {Incearca}

/* Model pentru rezolvarea unei probleme de tip (n,m) -
determinarea tuturor soluțiilor */

void incearcă()

{

for (*toate posibilitățile de selecție)

if (*selecție acceptabilă)

{

înregistrează-o ca și curentă /[5.4.4.a]*/

if (*soluția incompletă)

incearcă *pasul urmator

else

*evidențiază soluția;

*șterge înregistrarea curentă

}

} /*incearcă*/

/*-----*/

- Ca și în cazul anterior, acest model principal va fi concretizat în **două variante**.
- (1) În prima **variantă**:
 - Procedura Incearca1 are drept parametru de apel **numărul pasului curent** și realizează explorarea posibilităților în bucla interioară **FOR** [5.4.4.b].
 - Deoarece pentru evidențierea tuturor posibilităților în fiecare pas trebuie parcurs toate valorile lui $k=[1, m]$, ciclul **REPEAT** a fost înlocuit cu unul **FOR**.

{Rezolvarea unei probleme de tip (n,m). Determinarea tuturor
soluțiilor - Varianta 1}

PROCEDURE Incearca1(i: TipPas);

VAR posib: TipPosibilitate;

BEGIN

FOR posib:= 1 **TO** m **DO**

IF acceptabilă **THEN**

BEGIN

```

        *înregistrează-o ca și curentă;
        IF i<n THEN                                     [5.4.4.b]
            Incearca1(i+1)
        ELSE
            *afișează soluția;
            *șterge înregistrarea
        END
    END; {Incearca1}
}
-----
/* Rezolvarea unei probleme de tip (n,m). Determinarea
tuturor soluțiilor - Varianta 1 */

void incearca1(TipPas i)
{
    TipPosibilitate posib;
    for ( posib=1 pana_la m )
        if ( acceptabilă ) {
            *înregistrează-o ca și curentă;
            if ( i<n )                                     /*[5.4.4.b]*/
                incearca1(i+1)
            else
                *afișează soluția;
            *șterge înregistrarea
        }
    } /*incearca1*/
} /*-----*/

```

- (2) În cea de-a doua **variantă**:
 - Procedura Incearca2 are drept parametru de apel **o posibilitate de selecție**.
 - Construcția soluției se realizează apelând recursiv procedura succesiv pentru fiecare posibilitate în parte. [5.4.4.c].

{Rezolvarea unei probleme de tip (n,m). Determinarea tuturor soluțiilor - Varianta 2}

```

Procedura Incearca2(posib: TipPosibilitate);
BEGIN
    IF *acceptabilă THEN
        BEGIN
            *înregistrează-o ca și curentă;
            IF *soluție incompletă THEN
                BEGIN                                     [5.4.4.c]
                    Incearca2(Posibilitate1);
                    Incearca2(Posibilitate2);
                    ...
                    Incearca2(Posibilitatem);
                END
            ELSE
                *afișează soluția
            *șterge înregistrarea curentă
        END
    END

```



```

        END
    END; {Incearca2}
-----
{Rezolvarea unei probleme de tip (n,m). Determinarea tuturor
soluțiilor - Varianta 2}

void incearca2(TipPosibilitate posib)
{
    if ( *acceptabilă )
    {
        *înregistrează-o ca și curentă;
        if ( *soluție incompletă )
        {
            /*[5.4.4.c]*/
            incearca2(Posibilitate1);
            incearca2(Posibilitate2);
            ...
            incearca2(Posibilitatem);
        }
        else
            *afișează soluția
            *șterge înregistrarea curentă
        }
    } /*incearca2*/
} /*-----*/

```

- Pentru exemplificare, se prezintă generalizarea problemei celor 8 regine în vederea determinării tuturor soluțiilor [5.4.4.d].

{Problema celor 8 regine - determinarea tuturor soluțiilor}

```

PROGRAM OptRegine;
VAR i: integer;
    a: ARRAY[1..8] OF boolean;
    b: ARRAY[2..16] OF boolean;
    c: ARRAY[0..14] OF boolean;
    x: ARRAY[1..8] OF integer;

PROCEDURE Afisare;
VAR k: integer;
BEGIN
    FOR k:=1 TO 8 DO Write(x[k]);           [5.4.4.d]
    Writeln
END; {Afisare}
PROCEDURE Incearca(i: integer);
VAR j: integer;
BEGIN
    FOR j:=1 TO 8 DO
        IF a[j] AND b[i+j] AND c[i-j+7] THEN
            BEGIN
                x[i]:= j;
                a[j]:= false; b[i+j]:= false; c[i-j+7]:= false;
                IF i<8 THEN
                    Incearca(i+1)

```

```

        ELSE
            Afisare;
            a[j]:= true; b[i+j]:= true; c[i-j+7]:= true
        END{IF}
    END; {Incearca}

BEGIN {programul principal}
    FOR i:=1 TO 8 DO a[i]:= true;
    FOR i:=2 TO 16 DO b[i]:= true;
    FOR i:=0 TO 14 DO c[i]:= true;
    Incearca(1)
END.

-----
/* Problema celor 8 regine - determinarea tuturor soluțiilor
*/

#include <stdio.h>

typedef unsigned boolean;
#define true (1)
#define false (0)

int i;
boolean a[8];
boolean b[15];
boolean c[15];
int x[8];

void afisare()
{
    int k;
    for( k=1; k <=8; k++)
        printf("%i ", x[k-1]);                /*[5.4.4.d]*/
    printf("\n");
} /*afisare*/

void incearca(int i)
{
    int j;
    for( j=1; j <=8; j++)
        if (a[j-1] && b[i+j-2] && c[i-j+7])
        {
            x[i-1]=j;
            a[j-1]=false; b[i+j-2]=false; c[i-j+7]=false;
            if (i<8)
                incearca(i+1);
            else
                afisare();
            a[j-1]=true; b[i+j-2]=true; c[i-j+7]=true;
        } /*if*/
    } /*incearca*/

int main(int argc, const char* argv[])
{
    /*programul principal*/
    for( i=1; i <= 8; i++) a[i-1]=true;

```

```

for( i=2; i <=16; i++) b[i-2]=true;
for( i=0; i <=14; i++) c[i]=true;
incearca(1);
return 0;
}
/*-----*/

```

- Algoritmul prezentat anterior determină cele **92 de soluții** ale problemei celor 8 regine.
- De fapt, din cauza simetriei există doar **12 soluții distincte** care apar evidențiate în figura 5.4.4.a.

R1	R2	R3	R4	R5	R6	R7	R8
1	5	8	6	3	7	2	4
1	6	8	3	7	4	2	5
1	7	4	6	8	2	5	3
1	7	5	8	2	4	6	3
2	4	6	8	3	1	7	5
2	5	7	1	3	8	6	4
2	5	7	4	1	8	6	3
2	6	1	7	4	8	3	5
2	6	8	3	1	4	7	5
2	7	3	6	8	5	1	4
2	7	5	8	1	4	6	3
2	8	6	1	3	5	7	4

Fig.5.4.4.a. Soluțiile problemei celor 8 regine

5.4.5. Problema relațiilor stabile

- **Specificarea problemei relațiilor stabile:**
 - Se dau **două mulțimi disjuncte** A și B de cardinalitate egală n.
 - Se cere să se găsească o mulțime de n perechi $\langle a, b \rangle$ astfel încât $a \in A$ și $b \in B$, perechi care să satisfacă anumite **constrângeri**.
- Există multe **criterii** de alcătuire a perechilor care satisfac constrângeri, unul dintre ele este cel cunoscut sub denumirea de "**regula relației stabile**" ("**stable marriage rule**").
 - Se presupune că ia ființă un **ansamblu de dansuri** pentru care s-au înscris băieți și fete în număr egal (n).

- În general fiecare băiat și fiecare fată are **preferințe distincte** față de partenerul său.
- În baza acestor **preferințe** se constituie n **perechi** de dansatori.
- Dacă în această repartizare există un băiat și o fată care **nu formează o pereche, dar** care se **preferă reciproc** față de actualii parteneri repartizați lor, se spune că **repartizarea este instabilă**.
- Dacă **nu** există astfel de perechi, **repartizarea** se numește **stabilă**.
- Această situație se referă la o serie de probleme asemănătoare din viața de toate zilele în care atribuiri trebuie executate în raport cu anumite **preferințe**: căsătorie, alegerea unei facultăți de către un candidat, alegerea unei meserii, alegerea unui loc de muncă, etc.
- În cazul de față problema va fi oarecum **simplificată** întrucât se presupune că **lista de preferințe** este un **invariant** și că ea **nu se modifică** după o atribuire particulară.
- O modalitate de rezolvare a problemei este aceea de a se încerca alcătuirea perechilor de dansatori pe rând, până la epuizarea celor două grupuri.
- Pentru aflarea **tuturor** repartizărilor **stabile** se poate creiona rapid schița algoritmului de rezolvare întrucât aceasta este evident o **problemă** de tip (n, m) căreia îi trebuie determinate **toate soluțiile**.
- Pornind de la modelul [5.4.4.a], procedura **Incearca** reprezintă prima formă a algoritmului care determină perechile.
 - Ideea este de a lua pe rând **băieții** și de a le distribui partenere, căutarea desfășurându-se conform listei de preferințe a băieților [5.4.5.a].

{Problema relațiilor stabile - rafinarea 1}

```

PROCEDURE Incearca (b: TipBaiat);
VAR o: TipOrdin;
BEGIN
  FOR o:=1 TO n DO
    BEGIN
      *alege cea de-a o-a preferință a băiatului b
      IF *acceptabilă THEN
        BEGIN
          *înregistrează perechea;
          IF b nu este ultimul băiat THEN
            Incearca(Succ(b))
          ELSE
            *înregistrează setul stabil ;
            *șterge perechea
          END{IF}
        END{FOR}
      END; {Incearca}
    
```

{Problema relațiilor stabile - rafinarea 1}

```

void încearcă (TipBaiat b);
{
    TipOrdin o;
    for( o=1 ;o <=n; o++)
    {
        *alege cea de-a o-a preferință a băiatului b
        if( *acceptabilă )
            {
                /*[5.4.5.a]*/
                *înregistrează perechea;
                if( b nu este ultimul băiat )
                    încearcă(Succ(b));
                else
                    *înregistrează setul stabil ;
                    *șterge perechea
            }/*if*/
        }/*for*/
    }/*încearcă*/
}/*-----*/

```

- În continuare se vor preciza **structurile de date**.
 - Din rațiuni de claritate a programului se introduc **trei tipuri scalare** identice ca definiție, dar cu nume diferite [5.4.5.b].
- **Datele inițiale** sunt reprezentate prin două matrici care indică **preferințele băieților** respectiv **preferințele fetelor**.
 - PrefBaieti[b] reprezintă lista de preferințe a băiatului b, respectiv PrefBaieti[b,o] este fata care ocupă poziția o în lista băiatului b.
 - PrefFete[f] este lista de preferințe a fetei f, iar PrefFete[f,o] este băiatul care reprezintă cea de-a o-a alegere a sa (fig.5.4.5.a).

{Problema relațiilor stabile - definirea structurilor de date}

```

TYPE TipBaiat   = 1..n;
        TipFata    = 1..n;
        TipOrdin   = 1..n;

```

[5.4.5.b]

```

VAR PrefBaieti: ARRAY[TipBaiat,TipOrdin] OF TipFata;
        PrefFete: ARRAY[TipFata,TipOrdin] OF TipBaiat;

```

/*Problema relațiilor stabile - definirea structurilor de date */

```

typedef unsigned tip_baiat;
typedef unsigned tip_fata;
typedef unsigned tip_ordin;

```

/*[5.4.5.b]*/

```

tip_fata pref_baieti[n][n];
tip_baiat pref_fete[n][n];

```

```
/*-----*/
```

- Un exemplu de date de intrare pentru problema relațiilor stabile apare în figura 5.4.5.a
- **Rezultatul** execuției algoritmului va fi reprezentat în forma unui tablou de fete x , astfel încât $x[b]$ reprezintă **partenera băiatului** b .
- Pentru egalitatea dintre între băieți și fete se mai introduce un tablou de băieți y , astfel încât $y[f]$ precizează **partenerul fetei** f [5.4.5.c].

```
-----  
{Problema relațiilor stabile - definirea structurilor de  
date de ieșire}
```

```
VAR x: ARRAY[TipBaiat] OF TipFata;           [5.4.5.c]  
    y: ARRAY[TipFata] OF TipBaiat;
```

```
-----  
/* Problema relațiilor stabile - definirea structurilor de  
date de ieșire*/
```

```
tip_fata x[n];                               /*[5.4.5.c]*/  
tip_baiat y[n];  
/*-----*/
```

ordin	1	2	3	4	5	6	7	8
băiatul 1 selectează fata	7	2	6	5	1	3	8	4
2	4	3	2	6	8	1	7	5
3	3	2	4	1	8	5	7	6
4	3	8	4	2	5	6	7	1
5	8	3	4	5	6	1	7	2
6	8	7	5	2	4	3	1	6
7	2	4	6	3	1	7	5	8
8	6	1	4	2	7	5	3	8
fata 1 selectează băiatul	4	6	2	5	8	1	3	7
2	8	5	3	1	6	7	4	2
3	6	8	1	2	3	4	7	5
4	3	2	4	7	6	8	5	1
5	6	3	1	4	5	7	2	8
6	2	1	3	8	7	4	6	5
7	3	5	7	2	4	1	8	6
8	7	2	8	4	5	6	3	1

Fig.5.4.5.a. Date de intrare pentru determinarea relației stabile

- Este evident că **nu** sunt necesare ambele tablouri, unul putându-se determina din celălalt conform relațiilor: $x[y[f]] = f$ și $y[x[b]] = b$ care sunt valabile pentru toate perechile constituite.

- Deși valorile lui $y[f]$ pot fi determinate printr-o simplă baleare a lui x , totuși y va fi păstrat, pe de o parte din rațiuni de **echitate** iar pe de altă parte pentru **claritatea** și **eficiența** programului generat.
- Informațiile cuprinse în x și y servesc la determinarea **stabilității** setului de perechi de dansatori care se construiește.
 - Deoarece acest set este construit pas cu pas, generând câte o pereche căreia i se verifică imediat stabilitatea, x și y sunt necesari chiar înainte ca toate perechile să fie stabilite.
- Pentru a păstra **evidența candidaților** se utilizează două tablouri booleene auxiliare: BNeales respectiv FNealeasa [5.4.5.d].

{Problema relațiilor stabile - definirea unor structuri de date auxiliare}

```

BNeales: ARRAY[baiat] OF boolean;           [5.4.5.d]
FNealeasa: ARRAY[fata] OF boolean;
    BNeales[b]= false      {baiatul b are pereche}
    FNealeasa[f]= false    {fata f are pereche}

```

/* Problema relațiilor stabile - definirea unor structuri de date auxiliare */

```

boolean b_neales[baiat];                      /*[5.4.5.d]*/
boolean f_nealeasa[fata];
    b_neales[b]==false;    /*baiatul b are pereche*/
    f_nealeasa[f]==false;  /*fata f are pereche*/
/*-----*/

```

- Semnificația câmpurilor celor două tablouri apare în aceeași secvență.
 - După cum se observă, BNeales[b]= **false** implică că $x[b]$ este definit, deci băiatul b are pereche.
 - Iar FNealeasa = **false** implică că $y[f]$ este definit, deci fata f are pereche.
- Deoarece alcătuirea perechilor se face pornind de la băieți care sunt parcurși în ordine (bucla **FOR** din [5.4.5a]), faptul că un băiat k a fost selectat sau nu, se poate determina din valoarea curentă a lui b conform relației [5.4.5.e]:

NOT BNeales[k] = $k < b$ [5.4.5.e]

- Aceasta sugerează faptul că tabloul BNeales **nu** este necesar și în consecință se va utiliza un singur tablou, adică FNealeasa destinat fetelor.
- Condiția **acceptabilă* va fi rafinată prin intermediul condițiilor FNealeasa[f] și **stabilă*, unde condiția **stabilă* va fi rafinată ulterior.

- Aplicând aceste dezvoltări se obține rafinarea [5.4.5.f] .

{Problema relațiilor stabile - rafinarea 2}

```

PROCEDURE Incearca(b: TipBaiat);
  VAR o: TipOrdin; f: TipFata;
  BEGIN
    FOR o:=1 TO n DO
      BEGIN
        f:= PrefBaieti[b,o];
        IF FNealeasa[f] AND *stabilă THEN          [5.4.5.f]
          BEGIN
            x[b]:= f; y[f]:= b; FNealeasa[f]:= false
            IF b<n THEN
              Incearca(Succ(b))
            ELSE
              *înregistrează setul stabil;
              FNealeasa[f]:= true
            END
          END; {Incearca}
        
```

/* Problema relațiilor stabile - rafinarea 2 */

```

void incearca(tip_baiat b);
{
  tip_ordin o;
  tip_fata f;
  for(o=1 ;o<=n; o++)
  {
    f=pref_baieti[b,o];
    if( f_nealeasa[f] && *stabila )          /*[5.4.5.f]*/
    {
      x[b]=f; y[f]=b;
      f_nealeasa[f]=false
      if( b<n )
        incearca(succ(b));
      else
        *înregistreaza setul stabil;
        f_nealeasa[f]=true
    }
  } /*incearca*/
}
/*-----*/

```

- Elementul esențial în acest context îl constituie **determinarea stabilității**.
 - O primă observație se referă la faptul că **stabilitatea** se stabilește **prin definiție** din **compararea ordinelor de preferințe**.
- Ordinul unui băiat sau al unei fete **nu** stă direct la dispoziție din datele utilizate până în acest moment. El poate fi determinat printr-o căutare relativ costisitoare în tablourile PrefBaieti respectiv PrefFete.

- Deoarece în determinarea stabilității **stabilirea ordinului de preferință** este o operație foarte frecventă, pentru a o face cât mai eficientă se introduc două noi **structuri de date** [5.4.5.g].
 - Matricea OFB în care $OFB[b, f]$ reprezintă ordinul de preferință pe care îl are fata f în lista băiatului b .
 - Matricea OBF în care $OBF[f, b]$ reprezintă prioritatea băiatului b în lista fetei f .
- Este evident faptul că cele două matrici sunt **invariante** și pot fi ușor determinate din matricele inițiale PrefBaieti și PrefFete.

{Problema relațiilor stabile - definirea unor structuri de date auxiliare}

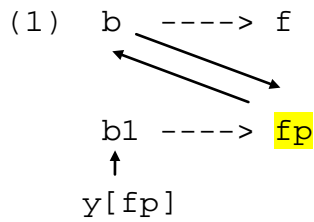
OFB: **ARRAY**[TipBaiat,TipFata] **OF** TipOrdin; {ordinul de preferință pe care îl au fetele în lista băieților} [5.4.5.g]

OBF: **ARRAY**[TipFata,TipBaiat] **OF** TipOrdin; {ordinul de preferință pe care îl au băieții în lista fetelor}

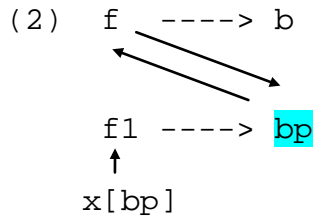
/* Problema relațiilor stabile - definirea unor structuri de date auxiliare */

```
tip_ordin ofb[n][n]; /*ordinul de
    preferință pe care îl au fetele în lista băieților*/
                                /*[5.4.5.g]*/
tip_ordin obf[n][n]; /*ordinul de
    preferință pe care îl au băieții în lista fetelor*/
/*-----*/
```

- În continuare **stabilitatea** se determină în raport strict cu **definiția sa inițială**.
 - Se reamintește faptul că în cadrul algoritmului se încearcă alcătuirea perechilor, selectând la un moment dat băiatul $b \in [1, n]$ și fata f , unde $f = \text{PrefBaieti}[b, o]$ adică f este cea de-a o -a clasată în lista de preferințe a lui b .
- În general, se presupune că stabilitatea este **predominantă** și în continuare se încearcă găsirea **surselor de perturbație** posibile.
- În acest sens există **două posibilități**:
 - **Sursa (1)** Presupunând că băiatului b i-a fost repartizată fata f , ar putea exista o fată f_p , preferată lui f de către b (f fiind perechea repartizată lui b), care la rândul ei preferă pe b perechii sale actuale b_1 [5.4.5.h (1)].
 - **Sursa (2)** Presupunând ca fata f a fost repartizată băiatului b , ar putea exista un băiat b_p preferat lui b de către f (f fiind perechea repartizată lui b), băiat care la rândul său preferă pe f perechii f_1 repartizate lui [5.4.5.h (2)].



[5.4.5.h]



- Inițial se urmărește **sursa** (1) de perturbație în vederea determinării **instabilității**:
 - Pentru toate fetele fp care au ordinul de preferință mai mic în lista lui b (adică sunt mai preferate) decât fata f repartizată lui, se verifică dacă **nu** cumva vreuna îl preferă mai mult pe b decât perechea ce i-a fost repartizată.
 - Pentru aceasta se compară ordinele $OFB[fp, b]$ și $OFB[fp, y[fp]]$ pentru toate fetele fp preferate lui f de către b , adică pentru toate $fp = PrefBaieti[b, i]$ astfel încât $1 \leq i < o$, unde o este ordinul de preferință al fetei f repartizate lui b .
 - De fapt, toate aceste fete fp sunt **deja distribuite** în perechi, deoarece dacă oricare dintre ele ar fi fost nedistribuită, ar fi fost repartizată lui b anterior.
 - Se face precizarea că un **ordin de preferință** cu valoare mai **mare** înseamnă o preferință mai **slabă**.
 - Rezultatul comparației este variabila booleană s care materializează **stabilitatea**. Atâta vreme cât s rămâne **true** relația este **stabilă**.
 - Procesul descris poate fi formulat printr-o simplă căutare liniară [5.4.5.i].

{Verificarea sursei 1 de perturbație}

```

s:= true; i:= 1;
WHILE (i<o) AND s DO
  BEGIN
    fp:= PrefBaieti[b,i]; i:= i+1;
    IF NOT FNealeasa[fp] THEN
      s:= OFB[fp,b]>OFB[fp,y[fp]]
  END;

```

[5.4.5.i]

/*Verificarea sursei 1 de perturbație*/

```

s=true; i=1;
while ((i<o) && s)                                /*[5.4.5.i]*/
{
    fp=pref_baieti[b,i]; i++;
    if (!f_nealeasa[fp])
        s=obf[fp,b]>obf[fp,y[fp]]?1:0;
}
/*-----*/

```

- În continuare se urmărește **sursa** (2) de perturbație:
 - Pentru toți băieții bp care au ordinul de preferință în lista fetei f mai mic ca și ordinul băiatului b căruia i-a fost repartizată, se verifică dacă nu cumva vreunul o preferă mai mult pe f decât pe perechea lui actuală.
 - În acest scop sunt investigați toți băieții preferați $bp = PrefFete[f, i]$ pentru $1 \leq i < OBF[f, b]$.
 - În mod analog ca și la cealaltă sursă de perturbație, este necesară compararea ordinele $OBF[bp, f]$ și $OBF[bp, x[bp]]$.
 - Vor trebui însă evitate comparațiile care se referă la băieții bp cărora încă **nu** le-au fost repartizate fete.
 - Testul care realizează acest obiectiv este $bp < b$, deoarece tuturor băieților care-l preced pe b li s-au repartizat deja perechi.
 - Rezultatul comparației este tot variabila booleană s care atâta vreme cât rămâne **true** relația este **stabilă**.
 - Secvența de cod care implementează verificarea celei de-a doua surse de perturbație apare în [5.4.5.j].

{Verificarea sursei 2 de perturbație}

```

i:= 1; lim:= OBF[f,b];
WHILE (i<lim) AND s DO                                [5.4.5.j]
    BEGIN
        bp:= PrefFete[f,i]; i:= i+1;
        IF bp<b THEN
            s:= OBF[bp,f]>OBF[bp,x[bp]]
        END;

```

/*Verificarea sursei 2 de perturbație*/

```

i=1; lim=obf[f,b];
while ((i<lim) && s)                                /*[5.4.5.j]*/
{
    bp=pref_fete[f,i]; i++;
    if (bp<b)
        s=ofb[bp,f]>ofb[bp,x[bp]]?1:0;
}

```

```

}
/*-----*/

```

- Algoritmul complet pentru determinarea relațiilor stabile apare în [5.4.5.k].

{Problema relațiilor stabile - varianta finală}

```

PROGRAM RelatieStabila;
CONST n=8;
TYPE TipBaiat = 1..n; TipFata = 1..n; TipOrdin = 1..n;

VAR b: TipBaiat; f: TipFata; o: TipOrdin;
    PrefBaieti: ARRAY[TipBaiat,TipOrdin] OF TipFata;
    PrefFete: ARRAY[TipFata,TipOrdin] OF TipBaiat;
    OFB: ARRAY[TipBaiat,TipFata] OF TipOrdin;
    OBF: ARRAY[TipFata,TipBaiat] OF TipOrdin;
    x: ARRAY[TipBaiat] OF TipFata;
    y: ARRAY[TipFata] OF TipBaiat;
    FNealeasa: ARRAY[TipFata] OF boolean;

PROCEDURE Afisează;
    VAR b1: TipBaiat;
        ob,of:integer;
    BEGIN
        ob:= 0; of:= 0;
        FOR b1:=1 TO n DO
            BEGIN
                Write(x[b1]);
                ob:= ob+OFB[b1,x[b1]]; of:= of+OBF[x[b1],b1]
            END;
        Writeln(ob,of)
    END; {Afiseaza}

PROCEDURE Incearca(b: baiat);
    VAR o: TipOrdin; f: TipFata;

    FUNCTION Stabila: boolean;
        VAR bp: TipBaiat; fp: TipFata;
            i,lim: TipOrdin; s: boolean;
        BEGIN
            s:= true; i:= 1; {sursa (1)}
            WHILE (i<o) AND s DO
                BEGIN
                    fp:= PrefBaieti[b,i]; i:= i+1;
                    IF NOT FNealeasa[fp] THEN
                        s:= OBF[fp,b]>OBF[fp,y[fp]]
                    END;
                i:= 1; lim:= OBF[f,b]; {sursa (2)}
                WHILE (i<lim) AND s DO
                    BEGIN
                        bp:= PrefF[f,i]; i:= i+1;
                        IF bp<b THEN s:= OFB[bp,f]>OFB[bp,x[bp]]
                    END;
                Stabila:= s
            END; {Stabilă}

```

```

BEGIN {Incearcă}
  FOR o:=1 TO n DO
    BEGIN
      f:= PrefBaieti[b,o];
      IF FNealeasa[f] THEN
        IF Stabila THEN
          BEGIN
            x[b]:= f; y[f]:= b; FNealeasa[f]:= false;
            IF b<n THEN
              Incearca(Succ(b))
            ELSE
              Afisează;
            FNealeasa[f]:= true
          END
        END
      END;
    END;
  END; {Incearcă}

```

```

BEGIN {programul principal}
  FOR b:=1 TO n DO
    FOR o:=1 TO n DO
      BEGIN
        Read(PrefBaieti[b,o]);
        OFB[b,PrefBaieti[b,o]]:=o
      END;
    FOR f:=1 TO n DO
      FOR o:=1 TO n DO
        BEGIN
          Read(PrefFete[f,o]); OBF[f,PrefFete[f,o]]:=o
        END;
      FOR f:=1 TO n DO FNealeasa[f]:= true;
      Incearca(1)
    END.

```

 /* Problema relațiilor stabile - varianta finală */

```

enum {n =8};
typedef unsigned char tip_baiat;
typedef unsigned char tip_fata;
typedef unsigned char tip_ordin;

```

```

typedef int          tip_ordin
typedef unsigned boolean;
#define true (1)
#define false (0)

```

```

tip_baiat b;
tip_fata f;
tip_ordin o;
tip_fata pref_baieti[n][n];
tip_baiat pref_fete[n][n];
tip_ordin ofb[n][n];
tip_ordin obf[n][n];
tip_fata x[n];
tip_baiat y[n];

```

```
boolean f_nealeasa[n];
```

```
void afiseaza()
```

```
{
    tip_baiat b1;
    int ob, of;
    ob=0; of=0; /*[5.4.5.k]*/
    for(b1=1; b1<=n; b1++){
        printf("%i", x[b1-1]);
        ob=ob+ofb[b1][x[b1-1]-1];
        of=oif+obf[x[b1-1]][b1-1];
    }
    printf("%i%i\n", ob,oif);
}/*afiseaza*/
```

```
void incearca(tip_baiat b);
```

```
static boolean stabila(tip_ordin* o, tip_baiat* b, tip_fata* f)
```

```
{
    tip_baiat bp; tip_fata fp;
    tip_ordin i,lim; boolean s;
    boolean stabila_result;
    s=true; i=1; /*sursa 1*/
    while ((i<*o) && s){
        fp=pref_baieti[*b][i-1]; i++;
        if (! F_nealeasa[fp-1])
            s=obf[fp][*b-1]>obf[fp][y[fp-1]-1];
    }
    i=1; lim=obf[*f][*b-1]; /*sursa 2*/
    while ((i<lim) && s){
        bp=pref_fete[*f][i-1]; i++;
        if (bp<*b)
            s=ofb[bp][*f-1]>ofb[bp][x[bp-1]-1];
    }
    stabila_result=s;
    return stabila_result;
} /*stabila*/
```

```
void incearca(tip_baiat b)
```

```
{
    tip_ordin o; tip_fata f;
    /*incearca*/
    for(o=1; o<=n; o++){
        f=pref_baieti[b][o-1];
        if (f_nealeasa[f-1])
            if (stabila(&o, &b, &f)){
                x[b-1]=f; y[f-1]=b;
                f_nealeasa[f-1]=false;
                if (b<n)
                    incearca(b++);
                else
                    afiseaza();
                f_nealeasa[f-1]=true;
            }
    }
}
```

```

    }/*incearca*/

int main(int argc, const char* argv[])
{
    /*programul principal*/
    for( b=1; b <=n; b++)
        for(o=1; o<=n; o++)
        {
            scanf("%c", &pref_baieti[b][o-1]);
            ofb[b][pref_baieti[b][o-1]-1]=o;
        }
    for(f=1; f<=n; f++)
        for(o=1; o<=n; o++)
        {
            scanf("%c", &pref_fete[f][o-1]);
            obf[f][pref_fete[f][o-1]-1]=o;
        }
    for(f=1; f<=n; f++) f_nealeasa[f-1]=true;
    incearca(1);
    return 0;
}
/*-----*/

```

- Pentru datele de intrare din fig.5.4.5.a se obțin soluțiile stabile din figura 5.4.5.b.
- Programul are la bază un **algoritm cu revenire** din categoria celor tratați în acest paragraf.
- **Eficiența** sa depinde în primul rând de gradul de sofisticare al soluției de reducere a parcurgerii arborelui căutărilor.
 - Există și alte metode mai eficiente decât cea propusă [MW71].
- Algoritmii de genul celui prezentat, care generează **toate** soluțiile posibile ale unei probleme sub incidența anumitor **constrângeri**, sunt adesea utilizați în selectarea uneia sau mai multor **soluții optime** într-un sens precizat.
- În cazul de față poate interesa spre exemplu soluția **cea mai favorabilă pentru băieți**, pentru **fete**, sau pentru **toți**.
- În tabela din fig.5.4.5.b apar în coloane specifice **suma ordinelor tuturor fetelor** alese din listele de preferințe ale perechilor lor băieți (ob) și **suma ordinelor tuturor băieților** aleși din listele de preferințe ale pe fetelor (of) pentru fiecare soluție în parte.
- Acestea sume se calculează conform relațiilor [5.4.5.1].

$$ob := \sum_{b=1}^n OFB[b, x[b]] \qquad of := \sum_{b=1}^n OFB[x[b], b] \qquad [5.4.5.1]$$

	x1	x2	x3	x4	x5	x6	x7	x8	ob	of
soluția 1	7	4	3	8	1	5	2	6	16	32
2	2	4	3	8	1	5	7	6	22	27
3	2	4	3	1	7	5	8	6	31	20
4	6	4	3	8	1	5	7	2	26	22
5	6	4	3	1	7	5	8	2	35	15
6	6	3	4	8	1	5	7	2	29	20
7	6	3	4	1	7	5	8	2	38	13
8	3	6	4	8	1	5	7	2	34	18
9	3	6	4	1	7	5	8	2	43	11

Fig.5.4.5.b. Soluții ale programului RelatieStabila

- Soluția cu cea mai **mică** valoare pentru ob se numește **soluția stabilă optimă pentru băieți**
- Soluția cu cea mai **mică** valoare pentru of se numește **soluția stabilă optimă pentru fete**.
 - Datorită strategiei de lucru adoptate în algoritm este evident că se obține la început soluția optimă pentru băieți, iar cea pentru fete la sfârșit. În acest sens algoritmul favorizează băieții.
 - Acest lucru poate fi însă ușor **evitat** dacă se schimbă rolul băieților și al fetelor respectiv se interschimbă PrefFete cu PrefBaieti și OFB cu OBF.

5.4.6. Problema selecției optime

- Următorul exemplu de **algoritm cu revenire** reprezintă o **extensie** ale celor anterioare.
 - Într-o primă etapă utilizând principiul revenirii s-a determinat **o singură soluție** pentru o anumită problemă de tip (n, m) (turneul calului sau problema plasării celor 8 regine).
 - În a doua etapă s-a abordat problema aflării **tuturor soluțiilor** unei probleme tip (n, m) (generalizarea plasării celor 8 regine).
 - În a treia etapă s-a abordat problema aflării **tuturor soluțiilor afectate de unele constrângeri** ale unei probleme tip (n, m) (relația stabilă).
- În continuare se propune aflarea **soluției optime**.
 - În acest scop se utilizează următoarea **strategie**: se generează **toate soluțiile posibile** iar pe parcursul procesului de generare este reținută soluția **optimă** într-un anumit sens.

- Presupunând că **optimul** este definit în termenii unei funcții cu valori pozitive $f(s)$, algoritmul căutat derivă din modelul destinat aflării **tuturor soluțiilor** unei **probleme de tip** (n, m) [5.4.4.a] în care afirmația **evidențiază soluția se înlocuiește cu secvența* [5.4.6.a].

```
-----
IF f(soluție)>f(optim) THEN optim:= soluție    [5.4.6.a]
-----
```

- Variabila **optim** înregistrează cea mai bună soluție întâlnită până la momentul curent și ea trebuie inițializată în mod corespunzător.
- Pentru a **nu** fi recalculată de fiecare dată, valoarea lui $f(optim)$ se păstrează de regulă într-o variabilă auxiliară.
- Un **model** cu caracter general al aflării **soluției optime** este următorul.
 - Se consideră o mulțime de obiecte, numită **mulțime de bază**.
 - Se cere să se găsească **maniera optimă de selecție** a unei **submulțimi de obiecte** care sunt supuse unor **constrângeri**.
 - Selecționările care constituie **soluții acceptabile**, se construiesc în mod gradat, examinând obiectele individuale ale **mulțimii de bază**.
- Prima rafinare a procedurii **Incearca** [5.4.6.b], schițează algoritmul de investigare al obiectelor mulțimii în vederea selecției, potrivit soluției.
 - Procedura se apelează **recursiv** (pentru a investiga obiectul următor) până la parcurgerea tuturor obiectelor.
 - Se precizează că **examinarea unui obiect** numit **candidat**, se poate termina în două moduri:
 - (1) Fie cu **includerea** obiectului investigat în soluția curentă.
 - (2) Fie cu **excluderea** sa din soluția curentă.
 - Din acest motiv utilizarea instrucțiunilor repetitive **REPEAT** sau **FOR nu-și** are sensul, cele două situații specificându-se în mod explicit în schița de algoritm [5.4.6.b].

```
-----
{Determinarea selecției optime - schiță de algoritm}
-----
```

```
PROCEDURE Incearca(i: TipObiect);
BEGIN
  IF *incluziunea este acceptabilă THEN
  BEGIN
    *include cel de-al i-lea obiect;
    IF i<n THEN
      Incearca(i+1)
    ELSE
      *verifică optimalitatea;
  END IF;
END;
```

[5.4.6.b]

```

        *elimină al i-lea obiect
    END;
    IF *excluziunea este acceptabilă THEN
        IF i<n THEN
            Incearca(i+1)
        ELSE
            *verifică optimalitatea
        END;{Incearca}
    }
}
-----
/* Determinarea selecției optime - schiță de algoritm */

void incearca(tip_obiect i);
{
    if(*incluziunea este acceptabila)
    {
        *include cel de-al i-lea obiect;
        if( i<n )
            incearca(i+1);
        else
            *verifica optimalitatea;
        *elimina al i-lea obiect
    }
    if(*excluziunea este acceptabila)
    if( i<n )
        incearca(i+1);
    else
        *verifica optimalitatea
    }/*incearca*/
}/*-----*/

```

- Se presupune că obiectele mulțimii de bază sunt numerotate cu $1, 2, \dots, n$.
 - Pornind de la acest model se obțin 2^n mulțimi selectate posibile.
- Este necesar însă să fie introduse **criterii** corespunzătoare care **să reducă** cât mai substanțial **numărul candidaților investigați** având în vedere faptul că se caută de fapt **selecția optimă**.
 - Pentru lămurirea acestor aspecte, **modelul** anterior va fi **particularizat** după cum urmează.
- **Specificarea problemei** selecției optime:
 - Se consideră o **mulțime de bază** formată din n obiecte A_1, A_2, \dots, A_n .
 - Fiecare obiect A_i este caracterizat prin **greutatea** sa g_i și prin **valoarea** sa v_i .
 - **Se cere** să se genereze **setul optim de obiecte** constând dintr-o selecție a celor n obiecte, selecție care este afectată de una sau mai multe **constrângeri**.
 - Se definește drept **set optim** acel set care are **cea mai mare sumă a valorilor componentelor**.

- Se precizează că **constrângerea** se referă la **limitarea sumei greutăților componentelor**.
- Aceasta este bine cunoscută **problemă a călătorilor (knapsack problem)**, care își pregătesc bagajele selectând lucruri dintr-o mulțime n, astfel încât:
 - **Valoarea** lor (de întrebuințare) să fie **maximă**.
 - **Greutatea** lor totală să **nu** depășească o anumită **limită**.
- **Rezolvarea problemei** începe cu etapa stabilirii **structurilor de date** [5.4.6.c].

{Determinarea selecției optime - definirea structurilor de date}

```

TYPE TipIndice = 1..n;
      TipObiect = RECORD
        g,v: integer
      END;

```

```

VAR obiecte: ARRAY[TipIndice] OF TipObiect;      [5.4.6.c]
      limg,vtot,vmax: integer;
      s,sopt: SET OF TipIndice;

```

/* Determinarea selecției optime - definirea structurilor de date */

```

enum {n=...};
typedef unsigned tipindice;
typedef int *set;

```

```

typedef struct{
    int g,v;
}tip_obiect;

```

```

tip_obiect obiecte[n];                                /*[5.4.6.c]*/
int limg,vtot,vmax;
set s,sopt;
/*-----*/

```

- Variabila limg precizează **greutatea limită** impusă selecției.
- Variabila vtot precizează **valoarea totală** a celor n obiecte.
 - Ambele valori sunt **constante** pe parcursul întregului proces de selecție.
- Variabila s reprezintă **selecția curentă a obiectelor**, în care fiecare obiect este reprezentat prin numele său (indice).
- Variabila sopt memorează **selecția optimă curentă**.

- Variabila v_{max} memorează **valoarea** selecției optime curente s_{opt} .
- În continuare se vor preciza **criteriile de acceptare** ale unui obiect în selecția curentă.
 - (1) În ceea ce privește **includerea**:
 - Un obiect este selectabil dacă el se încadrează în **toleranța de greutate**.
 - (2) În ceea ce privește **excluderea**:
 - Criteriul de acceptare al excluderii (cel care permite continuarea construirii selecției curente), este acela conform **căruia valoarea potențială finală** v_p la care s-ar putea ajunge **fără** a **include** candidatul curent în selecție, **nu este mai mică** decât optimul întâlnit până în prezent.
 - **Neincluderea candidatului curent** reprezintă practic **excluderea** sa din selecție.
 - În cazul în care valoarea v_p curent determinată este mai **mică** decât optimul curent, procesul de căutare **poate** conduce și la alte soluții, **singur** însă **nu** la una **optimă**, deci căutarea în pasul curent **nu** este fructificabilă, ca atare este abandonată.
- În baza acestor **două** condiții se precizează **elementele relevante** care urmează să fie determinate în fiecare pas al procesului de selecție:
 - (1) **Greutatea totală** g_t a selecțiilor executate până în prezent.
 - Variabila g_t se inițializează cu valoarea 0, căreia i se adaugă de fiecare dată greutatea obiectului inclus în selecția curentă.
 - Condiția ***incluziunea este acceptabilă** este ca noua greutate totală g_t să **nu depășească** limita de greutate lim_g [5.4.6.d].
 - (2) **Valoarea potențială** v_p care mai poate fi atinsă de către selecția curentă s .
 - Inițial v_p ia valoarea maximă posibilă v_{tot} obținută prin însumarea valorilor tuturor celor n obiecte ale mulțimii.
 - La fiecare excludere din v_p se scade valoarea obiectului exclus.
 - Condiția ***excluziunea este acceptabilă** este adevărată dacă noua valoare a lui v_p rămâne mai mare ca v_{max} unde v_{max} este valoarea corespunzătoare **soluției optime** determinate până în prezent [5.4.6.e].
 - Explicația este următoarea: chiar eliminând din selecție obiectul curent, valoarea potențială la care s-ar mai putea

ajunge este mai mare decât cea a soluției optime curente, adică selecția curentă poate încă conduce la o soluție optimă.

- Entitățile precizate `gt` și `vp` sunt declarate ca parametri ai procedurii **Incearca**.

{rafinarea afirmației *incluziunea este acceptabilă}

`gt + obiecte[i].g <= limg` [5.4.6.d]

{rafinarea afirmației *excluziunea este acceptabilă}

`vp - obiecte[i].v > vmax` [5.4.6.e]

-
- Spre a evita reevaluarea periodică a diferenței `vp-obiecte[i].v`, aceasta se notează cu `vp1`.
 - Verificarea **optimalității** și **înregistrarea soluției optime** apare în secvența [5.4.6.f].

{verifică optimalitatea}

IF `vp>vmax` **THEN**

BEGIN {avem un nou optim care se înregistrează}

`sopt:= s; vmax:= vp` [5.4.6.f]

END;

-
- Ultima secvență este bazată pe raționamentul că valoarea potențială devine valoare efectivă, de îndată ce au fost tratate toate cele `n` obiecte.
 - **Programul integral** în variantele Pascal și C apare în [5.4.6.g].
 - În varianta Pascal implementarea simplă și elegantă a incluziunii și excluziunii se datorează utilizării tipului mulțime **SET** și a operatorilor specifici definiți în limbaj.
 - În varianta C operațiile cu mulțimi sunt implementate într-o manieră explicită în forma tipului **set** respectiv a a operatorilor **inset**(`int`, `set`), **setof**(`int`,...), **join**(`set`,`set`) și **difference**(`set`,`set`).

PROGRAM SelectieOptima;

CONST `n=10;`

TYPE `TipIndice = 1..n;`

`TipObiect = RECORD`

`g,v: integer`

END;

VAR `i: TipIndice;`

`obiecte: ARRAY[TipIndice] OF TipObiect;`

`limg,vtot,vmax: integer;`

`g1,g2,ratia: integer;`

`s,sopt: SET OF TipIndice;`

```
z: ARRAY[boolean] OF char; [5.4.6.g]
b: boolean;
```

```
PROCEDURE Incearca(i: TipIndice; gt, vp: integer);
VAR vp1: integer;
BEGIN {se încearcă incluziunea obiectului i}
  IF gt+obiect[i].g<=limg THEN
    BEGIN
      s:= s+[i];
      IF i<n THEN
        Incearca(i+1,gt+obiecte[i].g,vp)
      ELSE
        IF vp>vmax THEN
          BEGIN
            vmax:= vp;
            sopt:= s
          END;
        s:= s-[i]
      END;
    {se încearcă excluderea obiectului}
    vp1:= vp-obiecte[i].v;
    IF vp1>vmax THEN
      BEGIN
        IF i<n THEN
          Incearca(i+1,gt,vp1)
        ELSE
          BEGIN
            vmax:= vp1;
            sopt:= s
          END
        END
      END
    END;{Incearca}
```

```
BEGIN {programul principal}
  vtot:= 0;
  FOR i:=1 TO n DO
    WITH obiecte[i] DO
      BEGIN
        Read(g,v);
        vtot:= vtot+v
      END;
  Read(g1,g2,ratia);
  z[true]:= '*';
  z[false]:= ' ';
  Write(' Greutate ');
  FOR i:=1 TO n DO Write(' ',obiecte[i].g);
  Writeln; Write(' Valoare ');
  FOR i:=1 TO n DO Write(' ',obiecte[i].v);
  Writeln;
  REPEAT
    limg:= g1;
    vmax:= 0;
    s:= [];
    sopt:= [];
    Incearca(1,0,vtot);
```

```

        Write('          ',limg);
        FOR i:=1 TO n DO
            BEGIN
                b:= i IN sopt;
                Write('          ',z[b])
            END;
        Writeln;
        g1:= g1+ratia
    UNTIL g1>g2
END.
-----
/*-----*/
/* Selecția Optimă */

#include <limits.h>
#include <stdarg.h>
#include <stdlib.h>

enum { n =10};
typedef unsigned char tip_indice;

typedef unsigned int boolean;
#define true (1)
#define false (0)

typedef int* set;
#define eos INT_MIN

typedef struct{
    int g,v;
} tip_obiect;

tip_indice i;
tip_obiect obiecte[n];
int limg,vtot,vmax;
int g1,g2,ratia;
set s,sopt;
char z[2];
boolean b;

boolean inset(int, set);
set setof(int,...);
set join(set,set);
set difference(set,set);

void incearca(tip_indice i, int gt,int vp)
{
    int vp1;
    /*se incearca incluziunea obiectului i*/
    if (gt+obiecte[i-1].g<=limg){
        s=join(s, setof(1,i));
        if (i<n)
            incearca(i+1,gt+obiecte[i-1].g,vp);
        else
            if (vp>vmax){

```

```

        vmax=vp;
        sopt=s;
    }
    s=difference(s, setof(1,i));
}
/*se incarca excluderea obiectului*/
vp1=vp-obiecte[i-1].v;
if (vp1>vmax){
    if (i<n)
        incarca(i+1,gt,vp1);
    else{
        vmax=vp1;
        sopt=s;
    }
}
}/*incarca*/

int main(int argc, const char* argv[])
{
    /*programul principal*/
    vtot=0;
    for( i=1; i <=n; i++){
        tip_obiect* with =&obiecte[i-1];
        scanf("%i%i", &with->g,&with->v);
        vtot=vtot+with->v;
    }
    scanf("%i%i%i", &g1,&g2,&ratia);
    z[true]='*';
    z[false]=" ";
    printf("    Greutate  ");
    for( i=1; i <=n; i++)
        printf("%3i", obiecte[i-1].g);
    printf("\n");
    printf("    Valoare  ");
    for( i=1; i <=n; i++)
        printf("%3i", obiecte[i-1].v);
    printf("\n");
    do {
        limg=g1;
        vmax=0;
        s=setof(0);
        sopt=setof(0);
        incarca(1,0,vtot);
        printf("%3i", limg);
        for( i=1; i <=n; i++){
            b=inset(i, sopt);
            printf("%4c", z[b]);
        }
        printf("\n");
        g1=g1+ratia;
    } while (!(g1>g2));
    return 0;
}

/*-----Implementarea operațiilor cu mulțimi-----*/

```



```

boolean inset(int x, set s)
{
    int i=0;
    for (i=1; i<=s[0]; i++)
        if (s[i] ==x)
            return true;
    return false;
}

set setof(int x,...)
{
    va_list ap;
    set s;
    int i=0,j=0,t,f;
    if ((s=malloc((x+1) * sizeof(int)))==NULL){
        printf("eroare!");
        exit(1);
    }
    s[0]=0;
    if (x){
        va_start(ap,x);
        for (i=1; i<=x; i++){
            t=va_arg(ap,int);
            f=1;
            for (j=1; j<i; j++)
                if ( t ==s[j])
                    f=0;
            if (f){
                s[i] =t;
                s[0]++;
            }
        }
        va_end(ap);
    }
    return s;
}

set join(set s1,set s2)
{
    int i=1;
    int j=1;
    set s;
    s=realloc(NULL,(s1[0]+s2[0]+1)*sizeof(int));
    s[0]=0;
    for (i =1; i<=s1[0]; i++){
        s[i]=s1[i];
        s[0]++;
    }
    for (j =1; j<=s2[0]; j++)
        if (inset(s2[j],s) ==false ){
            s[i++]=s2[j];
            s[0]++;
        }
    return s;
}

```

```

set difference(set s1, set s2)
{
    set s;
    int i;
    int j;
    s=realloc(NULL,(s1[0]+s2[0]+1)*sizeof(int));
    for (i=0; i<=s1[0]; i++)
        s[i]=s1[i];
    for (j =1; j<s2[0]; j++)
        if (!inset(s2[j],s)){
            s[0]++;
            s[i++]=s2[j];
        }
    return s;
}
/*-----*/

```

- În fig.5.4.6.c sunt listate rezultatele **execuției** programului cu **limita de greutate** cuprinsă în intervalul $10 \div 120$ cu ratia 10.
- **Algoritmii** care se bazează pe **tehnica revenirii** și care utilizează un **factor de limitare** care restrânge parcurgerea arborelui potențial de căutare, se mai numesc și algoritmi de tip "înlănțuie și limitează" ("**branch and bound algorithms**").

Greutate	10	11	12	13	14	15	16	17	18	19
Valoare	18	20	17	19	25	21	27	23	25	24
10	*									
20							*			
30					*		*			
40	*				*		*			
50	*	*		*			*			
60	*	*	*	*	*					
70	*	*			*		*		*	
80	*	*	*		*		*	*		
90	*	*			*		*		*	*
100	*	*		*	*		*	*	*	
110	*	*	*	*	*	*	*		*	
120	*	*			*	*	*	*	*	*

Fig.5.4.6.c. Rezultatele execuției programului SelectieOptima

5.5. Structuri de date recursive

5.5.1. Structuri de date statice și dinamice

- În cadrul capitolului 1 au fost prezentate **structurile de date fundamentale** respectiv tipurile nestructurate **enumerare**, **standard predefinite** și **subdomeniu** precum și tipurile structurate **tablou**, **articol** și **secvență**.
- Ele se numesc **fundamentale** deoarece:
 - Constituie elementele de bază cu ajutorul cărora se pot dezvolta structuri mai **complexe**.
 - Apar foarte frecvent în practică.
- Scopul **definirii tipurilor de date** urmat de **specificarea** faptului că anumite **variabile** concrete sunt de un anumit **tip**, este acela de:
 - (1) A fixa **domeniul valorilor** pe care le pot lua aceste variabile.
 - (2) A preciza **structura**, **dimensiunea** și **amplasarea** zonelor de **memorie** care le sunt asociate.
 - (3) A preciza **operatorii** care pot procesa aceste variabile.
- Întrucât toate aceste elemente sunt fixate de la început de către **compilator**, astfel de **variabile** și **structurile de date aferente lor** se numesc **statice**.
- În practica programării însă există multe probleme care presupun **structuri ale informației mai complicate**, a căror caracteristică principală este aceea că se **modifică structural** în timpul execuției programului.
 - Din acest motiv aceste structuri se numesc **structuri dinamice**.
- Este însă evident faptul, că la un anumit nivel de detaliere, componentele unor astfel de structuri sunt **structuri de date fundamentale**.
- În general, indiferent de limbajul de programare utilizat, o **structură de date statică** este aceea care ocupă **pe toată durata execuției programului** căruia îi aparține, o **zonă de memorie fixă**, de **volum constant**.
 - Memoria necesară unei **structuri statice** se rezervă în faza de **compilare** deci înainte de lansarea programului.
- Se impun câteva precizări:
 - (1) Se consideră **statice** acele **structuri de date** care au **volumul efectiv constant**.
 - (2) Se consideră tot **statice** acele **structuri de date** cu volum variabil pentru care programatorul poate aprecia o **margină superioară** a volumului și pentru care se alocă **volumul de memorie** corespunzător **cazului maxim** în faza de **compilare**.

- În contrast, o **structură de date dinamică** este aceea structură al cărei **volum de memorie nu poate fi cunoscut** în faza de compilare deoarece el se modifică funcție de maniera de **execuție** a **algoritmului**.
 - Cu alte cuvinte acest volum poate să **crească** sau să **descrească** în dependență de anumiți **factori** cunoscuți **numai** în timpul rulării.
 - În consecință **alocarea memoriei** la o **structură dinamică** are loc în timpul **execuției** programului.
- În general, orice **variabilă** declarată în partea de declarare a variabilelor, cu excepția celor de tip secvență, reprezintă o **structură statică**, pentru care se rezervă o **zonă de memorie constantă**.
 - Prin declararea unei **variabile statice** se precizează **numele** și **tipul** ei.
 - **Numele** variabilei este un identificator prin intermediul căruia programul, respectiv programatorul, poate face **referire** la această variabilă.
- Până în momentul de față, singurele structuri dinamice abordate au fost cele de tip **secvență**.
 - Datorită faptului că **secvențele** se presupun înregistrate în memorii externe, ele se consideră structuri dinamice **speciale** și **nu** fac obiectul prezentului paragraf.
- De fapt atât structurile **dinamice** cât și cele **statice** sunt formate în ultimă instanță din componente de **volum constant** care se încadrează într-unul din **tipurile cunoscute** și care sunt înregistrate integral în memoria centrală.
 - În cadrul **structurilor dinamice** aceste componente se numesc de regulă **noduri**.
- Natura **dinamică** a acestor structuri rezultă din faptul că **numărul nodurilor** se poate modifica pe durata rulării.
- Atât **structurile dinamice** cât și **nodurile** lor individuale se deosebesc de structurile statice prin faptul că ele **nu se declară** și în consecință **nu** se poate face referire la ele utilizând **numele** lor, deoarece practic **nu au nume**.
 - **Referirea** unor astfel de structuri se face cu ajutorul unor **variabile statice**, definite special în acest scop și care se numesc variabile **indicator (pointer)**.
- **Variabilele** referite în această manieră se numesc **variabile indicate**.

5.5.2. Tipul de date abstract indicator

5.5.2.1. Definirea TDA Indicator

- Utilizarea **structurilor de date dinamice** alături de alte aplicații speciale impun definirea unui **tip special** de variabile numite **variabile indicator**.
 - Valorile acestor variabile **nu** sunt **date efective** ci ele **precizează** (indică) **locații de memorie** care memorează **date efective**.
- Cu alte cuvinte, **valoarea** unei astfel de **variabile indicator** reprezintă o **referință** la o variabilă de un **anumit tip precizat**, numită **variabilă indicată**.
- Pentru a preciza natura acestor **variabile indicator**, s-a introdus un nou **tip de date abstract** și anume **tipul de date abstract indicator**.
- Descrierea de principiu a unui **tip de date abstract indicator** apare în [5.5.2.a].

TDA Indicator

Modelul matematic: Constă dintr-o mulțime de valori care indică adresele de memorie ale unor variabile numite indicate, aparținând unui tip specificat. Această mulțime de valori cuprinde și indicatorul vid care nu indică nici o variabilă.

Notății: p, q - variabile de TipIndicator;
 e - variabila de TipIndicat;
 b - valoare booleana. [5.5.2.a]

Operatori:

New(p) - procedură care alocă memorie pentru o variabilă indicată și plasează valoarea adresei de memorie (indicatorul) în variabila p ;

Dispose(p) - procedură care eliberează zona de memorie (locația) corespunzătoare variabilei indicate p ;

MemoreazăIndicator(p, q) - copiază indicatorul p în q ;

MemoreazăValoareIndicată(p, e) - copiază valoarea lui e în zona (locația) indicată de p ;

$e :=$ **FurnizeazăValoareIndicată**(p) - funcție care returnează valoarea memorată în zona (locația) indicată de p ;

$b :=$ **IndicatorIdentific**(p, q) - funcție booleană ce

returnează valoarea true dacă $p \equiv q$, adică cele două variabile indicator indică aceeași locație de memorie.

- Tipul de date abstract indicator poate fi implementat în mai multe moduri.
- În continuare se prezintă două astfel de modalități, una bazată pe **pointeri** și o a doua bazată pe **cursori**.

5.5.2.2. Implementarea TDA Indicator cu ajutorul pointerilor

- După cum se cunoaște, **variabilele pointer**, sunt variabile statice obișnuite care se declară ca orice altă variabilă.
 - Elementul particular al acestor variabile este faptul că ele se declară de tip **pointer**.
- Înainte de a preciza acest tip, se va clarifica **semnificația** variabilelor pointer.
 - **Valoarea** unei astfel de variabile este de fapt o **adresă de memorie** care poate fi adresa unei structuri dinamice sau a unei componente a ei.
 - În consecință, în limbaj de asamblare accesul la variabila indicată de o variabilă pointer se realizează prin **adresarea indirectă** a celei din urmă.
 - În limbajele de nivel superior același lucru se realizează prin atașarea unor caractere speciale numelui variabilei pointer în dependență de limbajul utilizat [5.5.2.2.a].

{Precizarea unei variabile indicate - varianta Pascal}

VariabilaIndicata \equiv VariabilaPointer[^] [5.5.2.2.a]

//Precizarea unei variabile indicate - varianta C

VariabilaIndicata \equiv *VariabilaPointer (în C) //[5.5.2.2.a]

- În plus, în limbajul C s-a definit operatorul **&** care se permite **determinarea adresei** unei variabile indicate și a fost dezvoltată o **aritmetică** specială a **pointerilor**.
- O **regulă** deosebit de importantă precizează că, spre deosebire de un limbaj de asamblare, în limbajele de nivel superior se stabilește o **legătură fixă** între orice **variabilă pointer** și **tipul variabilei indicate**.
 - O **variabilă pointer** dată se referă în mod **obligatoriu** la o **variabilă indicată** de un anumit tip.

- Această restricție mărește **siguranța** în programare și constituie o deosebire netă între **variabilele pointer** definite în limbajele de nivel superior și **adresele obișnuite** din limbajele de asamblare.
- Un **tip pointer** se definește precizând **tipul variabilei indicate** precedat de caracterul **"^"** în Pascal respectiv de caracterul **"*"** în limbajul C [5.5.2.2.b].

{Definirea unui tip pointer - varianta Pascal}

```
TYPE TipPointer = ^TipIndicat;      (Pascal)    [5.5.2.2.b]
```

//Definirea unui tip pointer - varianta C

```
tip_indicat  *variabila_pointer;    (C)          //[5.5.2.2.b]
```

- În limbajul C această restricție poate fi evitată utilizând tipul generic **void** care permite declararea unui **pointer generic** care **nu** are asociat un tip de date precis.

```
void *variabila_pointer;            [5.5.2.2.c]
```

- **Alocarea** sau **eliberarea memoriei** unei structuri dinamice în timpul rulării cade în sarcina **programatorului** și se realizează cu ajutorul unor **operatori standard** specifici dependenți de limbaj.
- Pentru exemplificare în secvențele următoare se prezintă implementarea **TDA Indicator** în limbajele Pascal respectiv C.
 - Implementările sunt realizate prin intermediul unor operatori definiți la nivelul limbajului.

{TDA Indicator - implementare Pascal}

```
TYPE TipPointer = ^TipIndicat;
VAR p,q: TipPointer;
    e: TipIndicator;      [5.5.2.2.d]
    b: boolean;
```

```

New(p);           {New(p)}
Dispose(p);       {Dispose(p)}
p:= q;            {MemoreazaIndicator(p,q)}
p^:= e;           {MemoreazaValoareIndicata(p,e)}
e:= p^;           {e:=FurnizeazaValoareIndicata(p)}
b:= p=q;          {b:=IndicatorIdentific(p,q)}
```

//TDA Indicator - implementare C

```
tip_indicat  *p,*q,e;
int b;                                              [5.5.2.2.e]
```

```
p=malloc(sizeof(tip_indicat));  {New(p)}
```

<code>free(p);</code>	<code>{Dispose(p)}</code>
<code>p= q;</code>	<code>{MemoreazaIndicator(p,q)}</code>
<code>*p= e;</code>	<code>{MemoreazaValoareIndicata(p,e)}</code>
<code>e= *p;</code>	<code>{e:=FurnizeazaValoareIndicata(p)}</code>
<code>b= (p==q);</code>	<code>{b:=IndicatorIdentific(p,q)}</code>

5.5.2.3. Implementarea TDA Indicator cu ajutorul cursorilor

- Dacă în limbajele care definesc **tipul pointer** (referință), **implementarea TDA Indicator** este imediată și realizată chiar la nivelul limbajului, în limbajele care **nu** dispun de această facilități sau în situații speciale, implementarea acestui tip de date se poate realiza cu ajutorul **cursorilor**.
- Un **cursor** este o **variabilă întreagă** utilizată pentru a indica o **locație** într-un **tablou** de **variabile indicate**.
 - Ca și metodă de conectare, un **cursor** este perfect **echivalent** cu un **pointer** cu deosebirea că el poate fi utilizat în plus și în limbajele care **nu** definesc tipul referință.
 - Interpretând valoarea unei **variabile întregi** ca și un **indice** într-un **tablou**, în mod efectiv acea variabilă va **indica** locația respectivă din tablou.
 - **Variabila indicată** este conținută în locația respectivă din **tablou**.
- Cu ajutorul acestei tehnici, pot fi implementate practic **toate structurile de date** care presupun **înlănțuiri**, după cum se va vedea în capitolele următoare.
 - Este însă evident faptul că, sarcina **gestionării zonei de memorie** care se alocă dinamic revine **exclusiv programatorului**, cu alte cuvinte programatorul trebuie să-și dezvolte proprii operatori de tip New - Dispose respectiv alloc-free.
 - De asemenea, programatorul trebuie să rezerve **static** un **spațiu de memorie** pentru **tabloul** utilizat drept suport pentru **alocarea dinamică a memoriei**.
- Un exemplu în acest sens apare în Capitolul 6 dedicat **structurii listă**.

5.5.2.4. Implementarea tipului indicator cu ajutorul referințelor

- Programarea orientată spre obiecte a impus noi dezvoltări ale posibilităților de acces la elementele cu care programatorul operează în cadrul unui program.

- Astfel limbajele orientate spre obiecte cum ar fi C++ sau JAVA definesc așa numitele **referințe** care de fapt sunt o formă de implementare a **tipului indicator**.
 - O referință apare ca fiind **similară** în multe privințe cu un pointer, dar de fapt **nu** este un pointer.
- O **referință** este un **alias** (un alt nume) pentru o **variabilă precizată**.
 - O **referință** este de fapt un **nume** care poate fi folosit în **locul** variabilei originale.
 - Deoarece este un alias și **nu** un pointer, **variabila** pentru care este declarată referința trebuie să fie **specificată** în momentul declarării referinței.
 - În plus spre deosebire de un pointer o **referință nu poate fi modificată** pentru a indica o altă variabilă.

-
- **Exemplul 5.5.2.4.** Considerând în limbajul C++ o variabilă declarată după cum urmează [5.5.2.4.a]:

```
long numar = 0; [ 5.5.2.4.a ]
```

- Se poate declara o **referință** pentru ea utilizând următoarea declarație [5.5.2.4.b] :

```
long& ref_numar = numar; // Declaraare referință [ 5.5.2.4.b ]
```

- În acest caz se poate spune că de fapt ref_numar este de tip "**referință la long**".
- Referința poate fi în continuare utilizată în locul variabilei originale. Spre exemplu atribuirea [5.5.2.4.c]

```
ref_numar += 13; [ 5.5.2.4.c ]
```

- Are ca efect incrementarea variabilei numar cu 13.
- Spre **deosebire** de referința anterior precizată, un pointer la aceeași variabilă poate fi declarat astfel [5.5.2.4.d] :

```
long* pointer_numar = &numar; [ 5.5.2.4.d ]
```

- Ceea ce permite incrementarea variabilei numar după cum urmează [5.5.2.4.e]:

```
*pointer_numar += 13; [ 5.5.2.4.e ]
```

- Există o distincție evidentă între utilizarea unui pointer și utilizarea unei referințe.

- **Pointerul** este evaluat la **fiecare utilizare** și în conformitate cu adresa pe care o conține la momentul respectiv este accesată **variabila indicată**.
- **Referința nu** este evaluată la fiecare utilizare. Odată **definită** ea **nu mai poate fi modificată**, lucru care nu este valabil și pentru pointer.
- **Referința** este complet **echivalentă** cu variabila la care se referă.
- La o analiză sumară referința **nu** pare a fi altceva decât o **notație alternativă** pentru o anumită **variabilă** deoarece ea se comportă în acest sens.
- În realitate adevăratele valențe ale acestui concept sunt puse în evidență în cazul utilizării **obiectelor**.
 - Astfel, în limbajele orientate spre programarea pe obiecte, în momentul în care se asignează obiecte variabilelor sau se transmit obiecte ca argumente pentru metode, se transmit de fapt **referințele** acelor obiecte și nu obiectele ca atare sau copii ale lor.
 - Acest lucru are o importanță cu totul particulară în dezvoltarea de aplicații orientate spre obiecte.
 - Spre exemplu în limbajul JAVA **nu** există pointeri expliți nici așa numita "aritmetică a pointerilor" ci doar referințe, fără ca acest lucru să restrângă libertatea de programare, ci dimpotrivă.

5.5.3. Structuri de date recursive

- În cadrul paragrafelor acestui capitol, până în prezent, **recursivitatea** a fost prezentată ca o **proprietate** specifică **algoritmilor**, implementată în forma unor proceduri sau funcții care se apelează pe ele însele.
- În cadrul acestui paragraf se va prezenta **extinderea** acestei proprietăți și asupra **tipurilor de date** în forma așa-numitelor "**structuri de date recursive**".
 - Prin analogie cu algoritmii, prin **structură de date recursivă** se înțelege o **structură** care are cel puțin o **componentă** de același tip ca și **structura însăși**.
 - Și în acest caz, definițiile unor astfel de tipuri de date pot fi recursive în mod **direct** sau în mod **indirect** (vezi &5.1).
- Un **exemplu** simplu de **structură recursivă** îl reprezintă **lista înlănțuită** a cărei definire formală apare în [5.5.3.a].

{Exemplu 1 de structură de date recursivă - structura listă înlănțuită}

```

TYPE TipLista = RECORD
    info: TipInfo;                                [5.5.3.a]
    urm: TipLista    (incorrect)
END;

```

/ Exemplu 1 de structură de date recursivă - structura listă înlănțuită */*

```

typedef struct {
    tipinfo info;                                /*[5.5.3.a]*/
    tiplista urm;    (incorrect*)
} tiplista;
/*-----*/

```

- Un alt exemplu clasic de **structură recursivă** este **arborele genealogic** al unei persoane, adică structura care precizează toate rudele directe ascendente ale persoanei în cauză.
 - O astfel de structură poate fi definită ca și un articol cu trei componente, prima reprezentând numele persoanei, celelalte două fiind arborii genealogici ai părinților.
- Aceasta se exprimă formal astfel [5.5.3.b]:

{Exemplu 2 de structură de date recursivă - arborele genealogic al unei persoane}

```

TYPE TipGenealogie = RECORD
    nume: string;                                [5.5.3.b]
    tata, mama: TipGenealogie    (incorrect)
END;

```

/ Exemplu 2 de structură de date recursivă - arborele genealogic al unei persoane */*

```

typedef struct {
    char* nume;                                /*[5.5.3.b]*/
    tipgenealogie tata, mama;    (incorrect*)
} tipgenealogie;
/*-----*/

```

- Datorită câmpurilor urm respectiv tata și mama , care au același tip ca și structura însăși, tipurile definite sunt **recursive**.
- Se face **precizarea** că **definițiile** de mai sus **nu sunt corecte**, deoarece se utilizează identificatorii TipLista repectiv TipGenealogie înainte de a fi **complet definiți**.
 - Cu alte cuvinte, încalcă **convenția** de a nu utiliza niciodata un **tip în curs de definire** ci doar după **finalizarea definirii sale**.
 - Acest neajuns va fi remediat ulterior.

- Se constată ușor că cele două tipuri definesc **structuri infinite**.
 - Aceasta este o consecință directă a **recursivității** respectiv a faptului că există câmpuri de tip **identic** cu cel al structurii complete.
 - Este însă evident faptul că în practică **nu** se pot prelucra **structuri infinite**.
- În realitate nici listele nici arborii genealogici **nu sunt infiniti**.
 - În cazul arborilor genealogici spre exemplu, urmărind suficient de departe ascendența oricărei persoane se ajunge la strămoși despre care **lipsește informația**.
- Ținând cont de aceasta, se va modifica definiția tipului TipGenealogie astfel:
 - Dacă se ajunge la o **persoană necunoscută** atunci structura va conține **un singur câmp** notat cu 'XX'.
 - În **orice alt caz** structura conține **trei câmpuri** conform definiției anterioare.
- În figura 5.5.3.a se poate urmări o structură de TipGenealogie conform definiției modificate.

PETRU	ȘTEFAN	ADAM	XX
	XX		
	MARIA	XX	
		EVA	XX

Fig.5.5.3.a. Exemplu de structură de date recursivă

- Cu privire la **tipurile de date recursive** se precizează în general că, pentru a evita structuri infinite, definiția tipului trebuie să conțină o **condiție**, de care depinde prezența efectivă a componentei (sau a componentelor) recursive.
 - Ca și în exemplul de mai sus, în anumite condiții, componentele având același tip ca și structura completă, pot să **lipsească**.
 - În accepțiunea acestor condiționări pot exista **structuri recursive finite**.
- **Structurile recursive** pot fi implementate în limbajele de programare avansate **numai** în forma unor **structuri dinamice**.

- Prin **definiție**, orice **componentă** care are **tipul identic cu structura completă**, se **înlocuiește** cu un **pointer** care indică acea componentă.
- În secvențele [5.5.3.c,d] apar pentru exemplificare definiții corecte ale structurii recursive TipGenealogie în variantă Pascal respectiv C.

{Structură de date recursivă - arborele genealogic al unei persoane - varianta Pascal}

```

TYPE PointerGenealogie = ^TipGenealogie;
      TipGenealogie = RECORD
                           nume: string;                [5.5.3.c]
                           tata,mama: PointerGenealogie
      END;

```

// Structură de date recursivă - arborele genealogic al unei persoane - varianta C}

```

struct genealogie {
    char *nume;                [5.5.3.d]
    struct genealogie *tata, *mama;
}

```

- După cum s-a precizat, în anumite condiții, o componentă poate să lipsească dintr-o structură recursivă.
- În acest scop, mulțimea valorilor oricărui **tip referință** se extinde cu **referința vidă** (pointer nul) care nu se referă la nici o variabilă indicată.
 - Acest pointer se notează cu NIL respectiv NULL, el aparținând prin definiție oricărui tip referință.
 - **Absența unei componente recursive** se va indica asignând pointerul referitor la această componentă cu **referința vidă**.
- Cu aceste precizări, **structura recursivă** din figura 5.5.3.a poate fi reprezentată ca în și în figura 5.5.3.b.

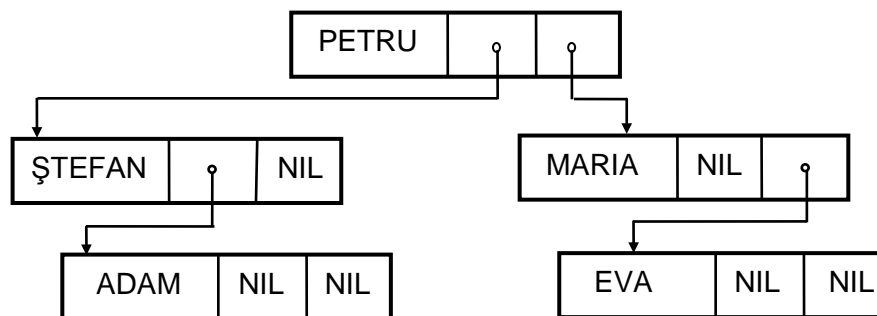


Fig.5.5.3.b. Implementarea unei structuri de date recursive

- Un alt exemplu de **structură recursivă** îl reprezintă **expresiile aritmetice**.
 - În acest caz **recursivitatea** rezultă din faptul că orice **expresie aritmetică** poate conține un **operand** care este la rândul său, o **expresie aritmetică** închisă între paranteze.
- În exemplul care urmează se va considera cazul simplu în care prin **expresie aritmetică** se înțelege fie:
 - Un **operand simplu** care este reprezentat ca un **identificator** format dintr-o singură literă.
 - O **expresie aritmetică** (operand) urmată de un **operator**, urmat de o **expresie aritmetică** (operand).
- Definirea unei astfel de structuri se poate realiza după cum se prezintă în secvența [5.5.3.e].

 {Structură de date recursivă - expresie aritmetică -
 varianta Pascal}

```

TYPE PointerExpresie= ^TipExpresie;
      TipExpresie = RECORD                                [5.5.3.e]
                    Operator: CHAR;
                    Operand1,Operand2: PointerExpresie
                    END;
  
```

 /* Structură de date recursivă - expresie aritmetică -
 varianta C */

```

typedef struct tipexpresie* pointerexpresie;
typedef struct {                                           /*[5.5.3.e]*/
    char operator_;
    pointerexpresie operand1,operand2;
}tipexpresie;

/*-----*/
  
```

- În cadrul acestei definiții se consideră că:

- Dacă în câmpul **operator** se găsește unul din caracterele '+', '-', '*' sau '/', semnificând o **operație aritmetică**, atunci celelalte două câmpuri conțin câte un **pointer** la o altă **expresie**.
- Dacă în câmpul **operator** se găsește o literă, semnificând un **operand**, atunci celelalte două câmpuri vor conține NIL.
- S-a arătat în &5.2.4, că orice **algoritm recursiv** poate fi înlocuit cu un **algoritm iterativ** echivalent.
- Această proprietate se păstrează și pentru **structurile de date recursive** și anume orice **structură recursivă** poate fi înlocuită cu o **structură secvențială**.
- Astfel, o definiție de tip recursiv de forma [5.5.3.f]:

{Structură de date recursivă}

```

TYPE IndicatorStructura = ^TipStructura
    TipStructura = RECORD
        continut: TipContinut;
        legatura: IndicatorStructura
    END;

```

- Este echivalentă și înlocuibilă imediat cu tipul secvențial [5.5.3.g]:

{Varianta iterativă a structurii de date recursive[5.5.3.f]}

```

TYPE TipStructura = FILE OF TipContinut

```

- Se observă că o **structură recursivă** se poate înlocui imediat cu una **secvențială** dacă și numai dacă numele tipului recursiv apare în **propria definiție recursivă** o **singură dată** la **sfârșitul** (sau la **începutul**) acesteia.
 - Această observație este valabilă și pentru **procedurile** sau **funcțiile recursive** (vezi &5.2.4).
- În cazul altor **structuri de date recursive** cu un grad mai ridicat de **complexitate** (de regulă bazate pe arbori de diferite ordine) se poate realiza **transformarea** într-o **structură secvențială** prin **traversarea** arborelui asociat structurii într-o manieră ordonată specifică (spre exemplu în **in**, **pre** sau **post** ordine respectiv **în adâncime** sau **prin cuprindere**).

5.6. Aplicații propuse

Aplicația 5.6.1

Se cere să se generalizeze algoritmul pentru evidențierea tuturor posibilităților de secționare a unui fir de lungime întreagă dată (&5.3.4) astfel încât să se găsească toate

variantele, inclusiv cea optimă, pentru secționarea unui fir în bucăți de lungimi $l_1, l_2, l_3, \dots, l_n$ date, în următoarele condiții:

- Nu există nici o restricție referitoare la numărul de bucăți de fiecare lungime.
- Se taie cel mult o bucată din fiecare lungime.
- Numărul de bucăți din fiecare lungime să difere cu cel mult o unitate.

Aplicația 5.6.2

Pornind de la algoritmul de determinare a tuturor soluțiilor de ieșire dintr-un labirint, se cere:

- Să se determine soluția optimă (drumul cel mai scurt până la ieșirea din labirint).
- Să se generalizeze algoritmul pentru determinarea tuturor soluțiilor de ieșire dintr-un labirint tridimensional.

Aplicația 5.6.3

Se cere să se realizeze o analiză comparată a performanțelor implementărilor recursive respectiv iterative ale următorilor algoritmi recursivi:

- Secționarea firului.
- Problema celor 8 regine.
- Turneul calului.

Analiza se va realiza pentru fiecare algoritm în parte cu evidențierea cantitativă în termeni de timp a diferențelor de performanță.

Aplicația 5.6.4

Între n orașe există o rețea de drumuri care permite ca dintr-un oraș să se ajungă în oricare din celelalte. Între două orașe există cel mult un drum direct de lungime cunoscută, iar timpul necesar pentru parcurgerea unui drum este proporțional cu lungimea sa. Se cere să se realizeze un program interactiv care determină traseul dintre două orașe precizate astfel încât timpul să fie minim.

Aplicația 5.6.5

Se consideră un set de n întrebări, fiecare având acordat un punctaj P_i . Se cere să se genereze toate chestionarele conținând un număr de întrebări cuprins între A și B având un punctaj total cuprins între C și D .

Aplicația 5.6.6

Se cere să se determine profilul performanței următorilor algoritmi recursivi:

- Turneul calului, pentru dimensiuni crescătoare ale tabelului, cu același punct de start;
- Problema celor 8 regine, pentru dimensiuni crescătoare ale tabelului.

Profilul va fi determinat atât pentru implementarea recursivă cât și pentru cea iterativă, execuțiile limitându-se la determinarea primei soluții.

Aplicația 5.6.7

Se cere să se realizeze analiza comparată a performanțelor implementărilor recursive respectiv iterative ale următorilor algoritmi recursivi:

- Problema relațiilor stabile.
- Problema selecției optime.

Analiza se va realiza pentru fiecare algoritm în parte cu evidențierea cantitativă în termeni de timp a diferențelor de performanță. Pentru creșterea acurateții determinării se vor utiliza mai multe seturi de date de intrare care să acopere diverse zone de comportament ale algoritmilor.