# 5. Recursivitate

## 5.1. Introducere

### 5.1.1. Definire. Iteraţie şi recursivitate

- O definiţie **recursivă** este acea **definiţie** care se referă la **un obiect** care **se defineşte ca parte a propriei sale definiri**.
    - Desigur o definiţie de genul *"o floare este o floare"* care poate reprezenta în poezie un univers întreg, în ştiinţă în general şi în matematică în special **nu** furnizează prea multe informaţii despre floare.
- O caracteristică foarte importantă a **recursivităţii** este aceea de a preciza o definiţie într-un **sens evolutiv**, care evită circularitatea.
    - Spre exemplu o **definiţie recursivă** este următoarea: *"Un buchet de flori este: (1) fie o floare, (2) fie o floare adăugată buchetului "*.
        - Afirmaţia (1) serveşte ca şi **condiţie iniţială**, indicând maniera de amorsare a definiţiei.
        - Afirmaţia (2) precizează **definirea recursivă (evolutivă) propriu-zisă**.
    - **Varianta iterativă** a aceleaşi definiţii este *"Un buchet de flori constă fie dintr-o floare, fie din două, fie din 3 flori, fie ... etc*".
        - După cum se observă, definiţia recursivă este **simplă** şi **elegantă** dar oarecum indirectă, în schimb definiţia iterativă este directă dar greoaie şi lipsită de eleganţă.
- Despre un **obiect** se spune că este **recursiv** dacă el **constă** sau **este definit prin el însuşi.**
    - Prin **definiţie** orice **obiect recursiv** implică **recursivitatea** ca şi proprietate **intrinsecă** a obiectului în cauză.
- **Recursivitatea** este utilizată cu multă eficienţă în **matematică**, spre exemplu în definirea numerelor naturale, a structurilor arbore sau a anumitor funcţii [5.1.1.a].

---

- **Numerele naturale**: [5.1.1.a]

    (1) 1 este un număr natural.

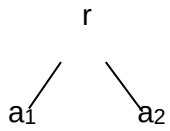
    (2) Succesorul unui număr natural este un număr natural.

- **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 $a_1$ şi $a_2$ sunt arbori binari, atunci structura




este un arbore binar ( reprezentat răsturnat).

- **Funcţia factorial** `n!` (definită pentru întregi pozitivi):

  (1) `0!=1`

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

---

- Puterea **recursivităţii** rezidă evident în posibilitatea de a defini un **set infinit de obiecte** printr-o **relaţie** sau un **set finit de relaţii,** caracteristică evidenţiată foarte bine de exemplele furnizate mai sus.
- Cunoscută ca şi un *concept fundamental* în **matematică**, recursivitatea a devenit şi o puternică facilitate de programare odată cu apariţia limbajelor de programare de nivel superior care o implementează ca şi **caracteristică intrinsecă** (ALGOL, PASCAL, C, JAVA etc.).
- În contextul programării, **recursivitatea** este strâns legată de **iteraţie** şi pentru a nu da naştere unor confuzii, se vor defini în continuare cele două concepte din punctul de vedere al **tehnicilor de programare**.
  - **Iteraţia** este **execuţia repetată** a unei porţiuni de program, până în momentul în care se îndeplineşte o **condiţie** respectiv atâta timp cât condiţia este îndeplinită.
    - Fiecare execuţie se duce pâna la **capăt**, se verifică îndeplinirea condiţiei şi în caz de răspuns nesatisfăcător se reia execuţia de la început.
    - Exemple clasice în acest sens sunt **structurile repetitive `while`, `repeat(do-while)`** şi **`for`** .
  - **Recursivitatea** presupune de asemenea **execuţia repetată** a unei porţiuni de program.
    - În **contrast** cu iteraţia însă, în cadrul recursivităţii, condiţia este verificată în **cursul** execuţiei programului şi nu la sfârşitul ei ca la iteraţie.
    - În caz de rezultat nesatisfăcător, întreaga porţiune de program este apelată din nou ca **subprogram (procedură sau funcţie) a ei însăşi**, în particular **ca procedură a porţiunii de program originale** care încă **nu** şi-a terminat execuţia.

- În momentul satisfacerii condiţiei de revenire, **se reia execuţia programului apelant** exact din punctul în care s-a apelat pe el însuşi.
- Acest lucru este valabil pentru **toate apelurile** anterioare satisfacerii condiţiei.

- Utilizarea **algoritmilor recursivi** este potrivită în situaţiile care presupun recursivitate respectiv **calcule**, **funcţii** sau **structuri de date** definite în **termeni recursivi**.

- În general, un **program recursiv** `P`, poate fi exprimat formal prin mulţimea `P` care constă din secvenţele de instrucţiuni $S_i, S_j$ care nu îl conţin pe `P` şi `P` însuşi. Se precizează ca fie $S_i$ fie $S_j$ pot fi secvenţa vidă . [5.1.1.b].

---

$$P \equiv P\ [S_i, P, S_j] \qquad [5.1.1.b]$$

---

## 5.1.2. Mecanismul implementării recursivităţii

- **Structurile de program** necesare şi suficiente pentru exprimarea recursivităţii sunt **procedurile**, **subrutinele** sau **funcţiile** care pot fi apelate prin nume.
  - Dacă o procedură `P` conţine o referinţă directă la ea însăşi, se spune că este **direct recursivă.**
  - Dacă `P` conţine o referinţă la o altă procedură `Q` care la rândul ei conţine o referinţă (directă sau indirectă) la `P`, se spune că `P` este **recursivă indirect** (figura 5.1.2.a).

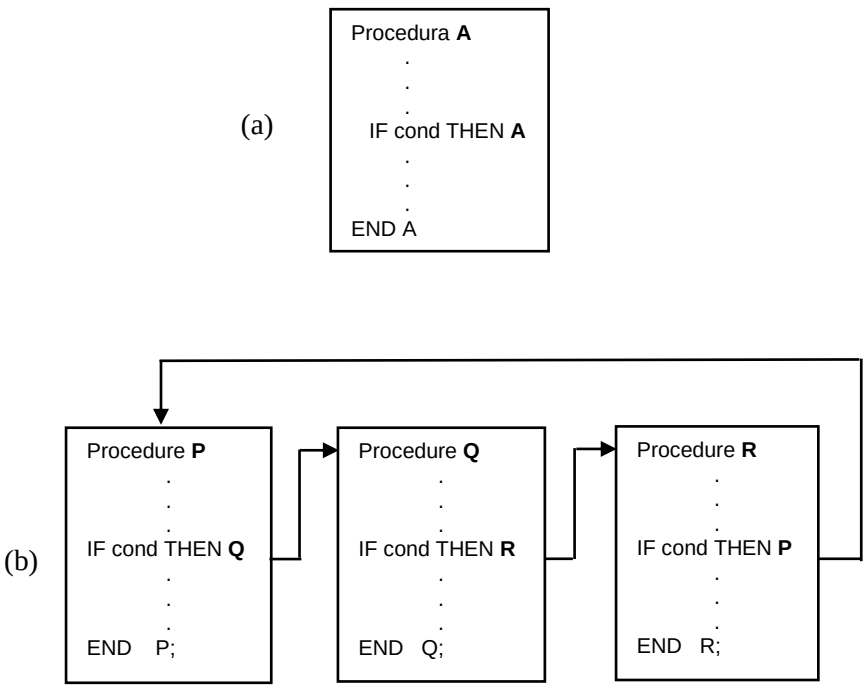



**Fig.5.1.2.a.** Recursivitate directă (a) şi recursivitate indirectă (b)

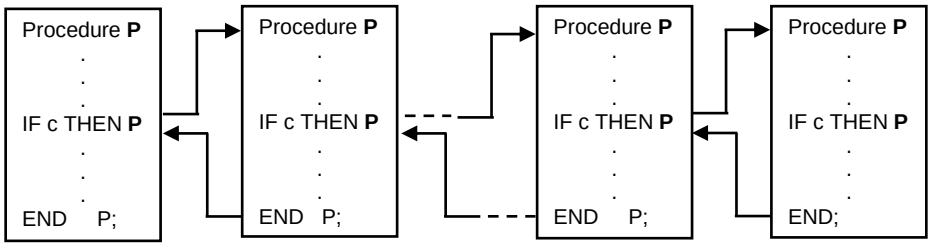



**Fig.5.1.2.b.** Model de execuţie al unei proceduri recursive

- Intuitiv, **execuţia unei proceduri recursive** este prezentată în figura 5.1.2.b.
    - Fiecare reprezentare asociată procedurii reprezintă o **instanţă de apel recursiv** a acesteia.
    - Instanţele se apelează unele pe altele până în momentul în care condiţia c devine falsă.
    - În acest moment, se finalizează execuţia instanţei curente şi **se revine pe lanţul apelurilor în ordine inversă**, continuând execuţia fiecărei instanţe din punctul de autoapel până la sfârşit, după cum indică săgeţile.
    - Suprapunând imaginar toate aceste figuri peste una singură, se obţine **modelul de execuţie** al unei **proceduri recursive**.
- De regulă, unei **proceduri** i se asociază un **set de obiecte locale procedurii** (variabile, constante, tipuri şi proceduri) care sunt definite local în procedură şi care nu există sau nu au înţeles în afara ei. Acest set de obiecte locale procedurii include şi **parametrii instanţei de apel.**
    - Mulţimea tuturor acestor obiecte constituie aşa 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**.
- Deşi variabilele din acest context, numit **context curent**, au acelaşi **nume** ca şi cele corespunzătoare lor din instanţa anterioară a procedurii (în calitate de program apelant), ele au valori **distincte** şi orice **conflict** de nume este evitat prin **regula** care stabileşte **domeniul de existenţă 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**.
    - Aceaşi regulă este valabilă şi pentru **parametrii de apel** ai procedurii care sunt asociaţi prin definiţie setului de variabile, respectiv contextului apelului.

- Pe măsură ce se realizează apeluri recursive, **contextul** fiecărui apel este **salvat** în **stivă**, iar pe măsură ce execuţia unei instanţe s-a încheiat, se **restaurează** contextul instanţei apelante prin eliminarea contextului aflat în vârful stivei.
- Acest **mecanism de implementare a recursivităţii** este ilustrat în figura 5.1.2.c.

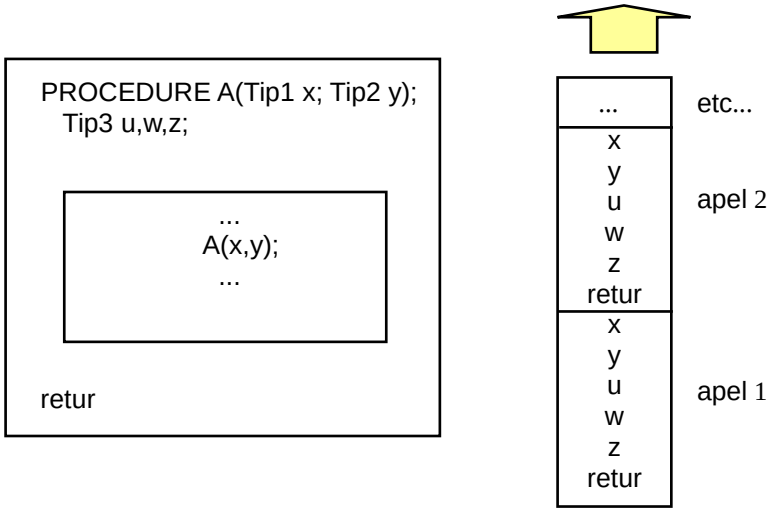



**Fig.5.1.2.c.** Mecanism pentru implementarea recursivităţii

- În legătură cu acest **mecanism** se pot face câteva observaţii:
  - (1) În cadrul apelurilor recursive **nu** apar nici un fel de probleme referitoare la **transmiterea parametrilor** interinstanţe de apel, dacă aceşti parametri se transmit prin **valoare**, ei fiind salvaţi 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 instanţele de apel, este necesară **declararea suplimentară a unor parametri locali**, câte unul pentru fiecare parametru de acest tip, care să asigure retransmiterea valorilor calculate interinstanţe. Aceşti parametri trebuiesc reasignaţi cu valorile iniţiale înaintea revenirii din procedura recursivă [5.1.2.a].

```
-------------------------------------------------------------
void a(tip1 x, tip2* y, tip3* z)            /*[5.1.2.a]*/
    {
      tip1 u,w;
      tip2 y1;     /*parametru local pentru y*/
      tip3 z1;     /*parametru local pentru z*/
        ...
        a(u,&y1,&z1);
        ...
        *y=y1;     /*reasignare parametru iniţial y*/
        *z=z1;     /*reasignare parametru iniţial z*/
    } /*a*/
-------------------------------------------------------------
```

- (3) În cazul parametrilor de tip **referinţă** trebuie ţinut cont de faptul că în urma apelului recursiv al unei proceduri se realizează **automat** şi **atribuirea** parametrilor implicaţi în apel [5.1.2.b].

```
-------------------------------------------------------------
void a(tip_referinta r)                          /*[5.1.2.b]*/
  {
       ...
     a(r->drept);  /*simultan cu apelul se realizează şi
      atribuirea variabilei curente r cu valoarea r->drept*/
       ...
  } /*a*/
-------------------------------------------------------------
```

- Ca şi în cazul structurilor repetitive, procedurile recursive necesită evaluarea unei **condiţii 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 **condiţii** `c` care la un moment dat devine neadevărată.
- În baza acestei observaţii, un **algoritm recursiv** poate fi exprimat cu mai mare acurateţe 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 ≡ P [Si,IF c THEN P,Sj]                [5.1.2.c]
-------------------------------------------------------------
          P ≡ P [IF c THEN P,Sj]                   [5.1.2.d]
-------------------------------------------------------------
          P ≡ P [Si,IF c THEN P]                   [5.1.2.e]
-------------------------------------------------------------
```

- **Tehnica de bază** în demonstrarea **terminării** unei **iteraţii (repetiţii)** constă în:
  - (1) A **defini** o funcţie `f(x)` (unde `x` este o variabilă din program), astfel încât **condiţia** `f(x)<0` să implice satisfacerea condiţiei de terminare a iteraţiei.
  - (2) A **demonstra** că `f(x)` descreşte pe parcursul execuţiei iteraţiei respective.
- Într-o manieră similară, **terminarea** unui **algoritm recursiv** poate fi demonstrată, dovedind că `P` descreşte pe `f(x)`.
  - O modalitate particulară uzuală de a realiza acest lucru, este de a asocia lui `P` un **parametru**, spre exemplu `n` şi de a apela recursiv pe `P` utilizând pe `n-1` ca şi valoare de parametru.
  - Dacă se înlocuieşte **condiţia** `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) ≡ P [Si,IF n>0 THEN P(n-1),Sj]      [5.1.2.f]
-------------------------------------------------------------
```

- De regulă în aplicaţiile practice trebuie demonstrat **nu** numai că **adâncimea recursivităţii este finită** ci şi 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 şi **starea curentă** a programului, cu scopul de a fi **refăcută**, atunci când apelul curent al lui `P` se termină şi urmează să fie reluată instanţa apelantă.

- Neglijarea acestor aspecte poate avea drept consecinţă **depăşirea** spaţiului de memorie alocat stivei sistem, greşeală 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 funcţionare** al unei proceduri recursive.
- În cadrul programului se defineşte procedura recursivă **`Revers`:**
  - Procedura **`Revers`** citeşte câte un caracter şi îl afişează.
  - În acest scop în cadrul procedurii se declară variabila locală z de tip `char`.
  - Procedura verifică dacă caracterul citit este `blanc`:
    - În caz **negativ** procedura se autoapelează recursiv.
    - În caz **afirmativ** afişează caracterul `z` [5.1.3.1.a].

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

void revers()                                  /*[5.1.3.1.a]*/
 {
    char z;
    scanf("%c", &z);
    if (z !=" ")
       revers();
    printf("%c", z);
 } /*revers*/
-------------------------------------------------------------
```

- Execuţia procedurii **`revers`** conduce iniţial la afişarea caracterelor în ordinea în care sunt citite până la apriţia primului blanc.
  - Fiecare autoapel presupune salvarea în stiva sistemului a **contextului apelului** care în situaţia de faţă constă doar din variabila locală `z`.
- Apariţia primului caracter blanc presupune suprimarea apelurilor recursive ale procedurii declanşând continuarea execuţiei procedurii, până la terminarea sa pentru fiecare din apelurile anterioare.

- În cazul de faţă, aceasta presupune afişarea caracterului memorat în variabila `z`.

- Acest şir de reveniri va produce la început afişarea unui blanc, după care sunt afişate caracterele în **ordinea inversă** citirii lor.
  - Acest lucru se întâmplă deoarece fiecare **terminare** a execuţiei 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 acelaşi cuvânt **scris invers**.

### 5.1.3.2. Algoritm pentru traversare a unei liste înlănţuite

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

```
-------------------------------------------------------------
/* Traversarea recursivă a unei liste înlănţuite - varianta
C */

typedef struct tip_nod* tip_lista;        /*[5.1.3.2.a]*/
typedef struct
{
  int data;
  tip_lista urm;
} tip_nod;

void traversare(tip_lista p)
  {
    if(p!=0)
    {
/*[1]*/ prelucrare(p->data); /*[1]*/ traversare(p->urm);
/*[2]*/ traversare(p->urm);  /*[2]*/ prelucrare(p->data);
    }
  } /*traversare*/
-------------------------------------------------------------
```

- Algoritmul de traversare **nu** numai că este foarte **simplu** şi foarte **elegant**, dar prin simpla **inversare** a instrucţiunilor **`prelucrare`** şi **`traversare`** (`[1]` cu `[2]`) se obţine 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 poziţia curentă până la sfârşitul listei.
  - După ce s-a realizat această traversare se prelucrează informaţia din nodul curent.
- **Consecinţa** execuţiei algoritmului:

- Informaţia conţinută într-o anumită poziţie `p` este prelucrată **numai** după ce au fost prelucrate **toate** informaţiile 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 şi 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).

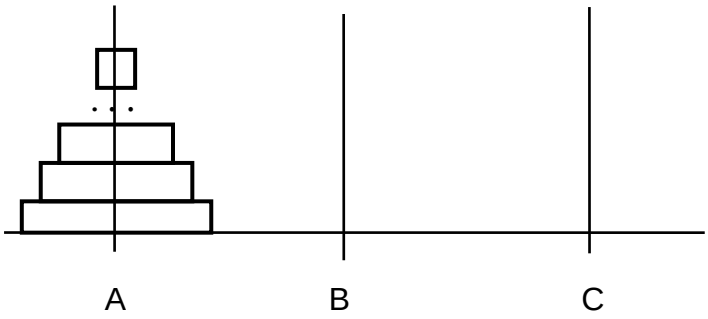



**Fig.5.1.3.3.a.** Model pentru problema Turnurilor din Hanoi

  - Se cere să se **mute** discurile pe vergeaua C utilizând ca şi auxiliar vergeaua B.
  - Se va respecta următoarea **restricţie**:
    - **Nu** se aşează niciodată un disc mai mare peste un disc mai mic.
- Problema pare simplă, dar rezolvarea ei cere multă, răbdare, acurateţe şi un volum de timp care creşte exponenţial odată cu `n`.
  - O abordare **recursivă** însă reprezintă o soluţie simplă şi 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 condiţiile impuse presupune următorii paşi :
    - (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 acelaşi 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.

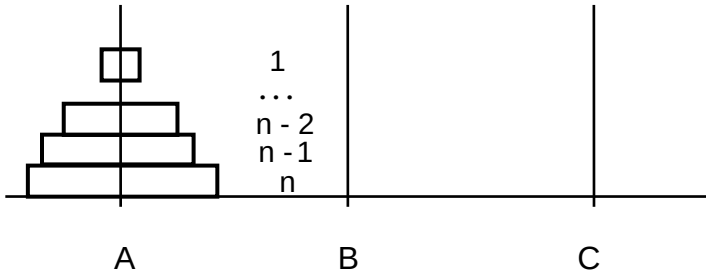



**Fig.5.1.3.3.b**. Model generalizat al problemei Turnurilor din Hanoi

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

```
-------------------------------------------------------------
/* Turnurile din Hanoi - varianta C */


void muta_disc(tip_vergea x, tip_vergea y) /*[5.1.3.3a]*/
/*mută discul dela vergeaua x la vergeaua y*/

  {
     *muta discul  de la x la y;
  } /*muta_disc*/

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

  { /*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);
        } /*else*/
  } /*turnuri_hanoi*/
-------------------------------------------------------------
```

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

## 5.2. Utilizarea recursivităţii

### 5.2.1. Cazul general de utilizare a recursivităţii

- **Algoritmii recursivi** sunt potriviţi a fi utilizaţi 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 recursivităţii în anumite situaţii nepotrivite, coroborată cu regia relativ ridicată a implementării şi execuţiei 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 construcţie a unui program recursiv pornind de la **modelul generic recursiv** [5.2.1.a] derivat din modelul [5.1.2.d].

---

```
P ≡ IF c THEN P[Si,P]                                    [5.2.1.a]
```

---

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

---

```
i = 0,1,2,3,4,5,...
fi = 1,1,2,6,24,120,...                                  [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].

---

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

---

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

---

```
Si ≡ (i=i+1; f=i*f;)                                     [5.2.1.d]
```

---

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

---

```
P ≡ IF i<n THEN (i=i+1; f=i*f; P)                        [5.2.1.e]
```

---

```
i= 0;  f= 1;  P;                                         [5.2.1.f]
```

---

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

---

```
/* Calculul factorialului - Varianta 1 C */

void p()                                          /*[5.2.1.g]*/
  {
    if(i<n){
       i++;
       f*=i;
       p();
    }
  }   /*p*/
```

---

- După cum s-a precizat, algoritmii recursivi se recomandă a fi utilizaţi cu precădere pentru implementarea unor probleme care se pot defini în **manieră recursivă**.
    - Cu toate acestea, **recursivitatea** poate fi utilizată şi 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 C */
/*apelul iniţial: cauta(1)*/

int n; tip_element x;                             /*[5.2.1.h]*/
tip_element *a;

int cauta(int i)
 {
    int cauta_result;
    if(i>n)
         cauta_result=0;
      else
        if(a[i-1]==x)
            cauta_result=i;
          else
            cauta_result=cauta(i+1);
    return cauta_result;
  }  /*cauta*/
```

---

- Este evident faptul că o astfel de abordare este posibilă, dar ea este **forţată** şi **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 funcţie, din următoarele raţiuni:

- **Funcţia** poate fi utilizată direct ca şi constituent al unei expresii.
- **Funcţiei** 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 funcţiei [5.2.2.a].

```
-------------------------------------------------------------
/*Funcţie recursivă pentru calculul factorialului - varianta
C*/

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

- Secvenţa în cauză care a fost redactată pentru calculul factorialului ca şi **întreg**, este **limitată** din punctul de vedere al dimensiunii maxime a lui `n` din raţiuni 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 funcţiei **`fact`** pentru `n = 4`.

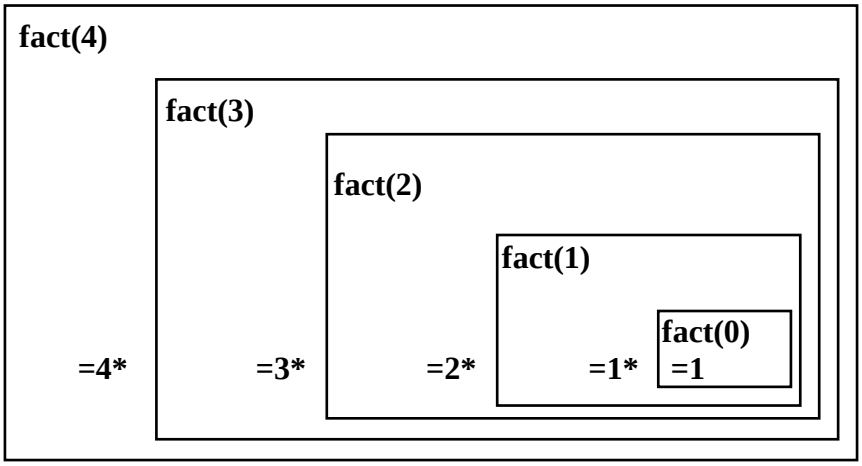



**Fig.5.2.2.a.** Apelurile ale funcţiei recursive **`fact`** pentru `n = 4`

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

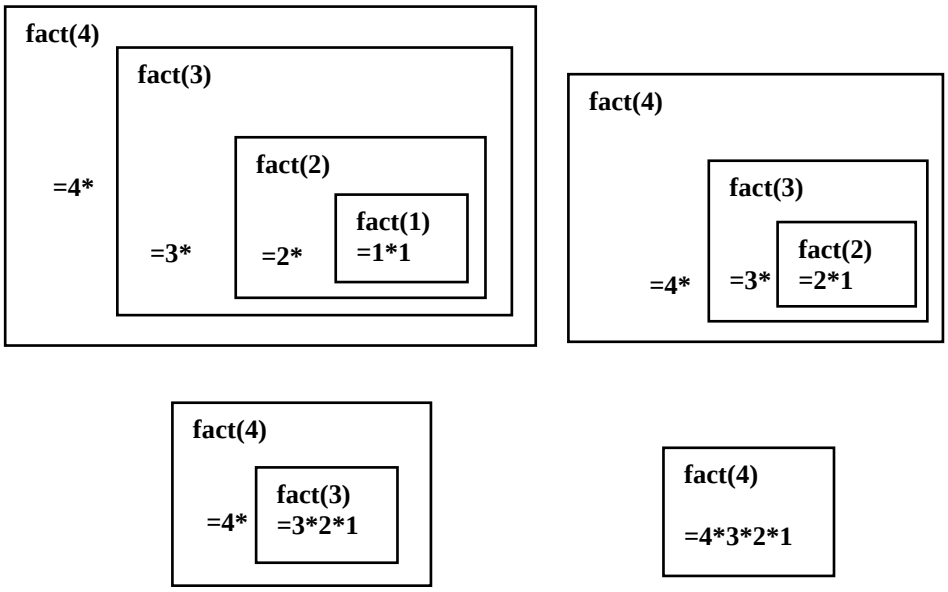



**Fig.5.2.2.b.** Rezolvarea apelurilor funcţiei recursive `Fact` pentru `n = 4`

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

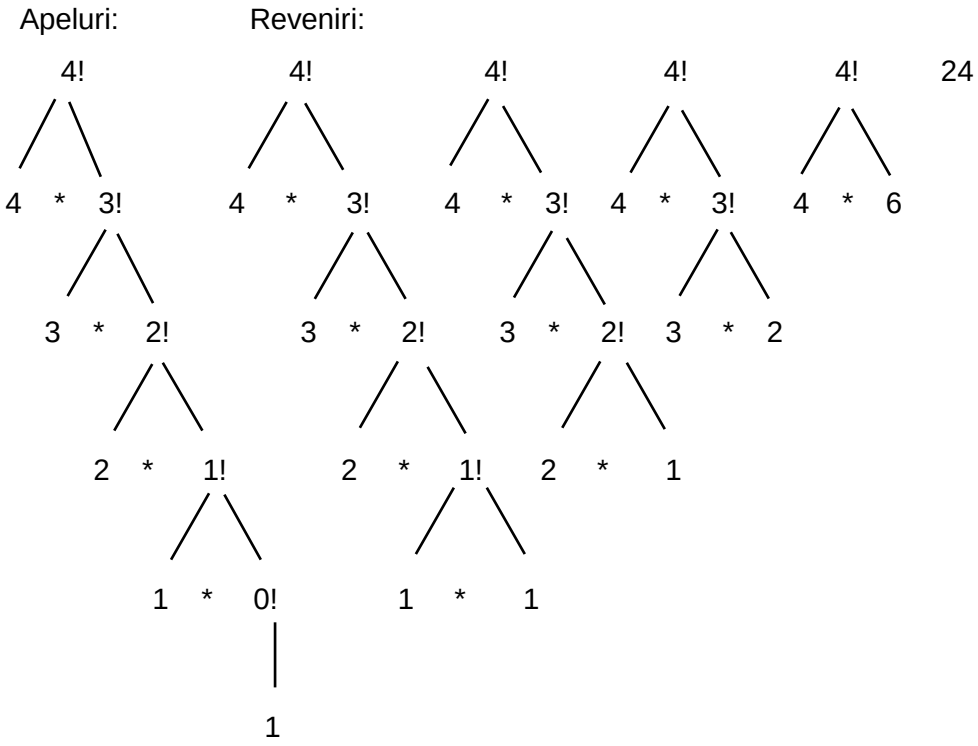



**Fig.5.2.2.c.** Arborele de apeluri al funcţiei `Fact(4)`

- În acest caz fiind vorba despre despre o funcţie cu **un singur apel recursiv**, **arborele de apeluri** are o structură de **listă liniară**.

- Fiecare apel adaugă un element la sfârşitul acestei liste, iar rezolvarea apelurilor are drept consecinţă reducerea succesivă a listei de la sfârşit spre început.

- După cum se observă, pentru a calcula factorialul de ordinul `n` sunt necesare `n+1` apeluri ale funcţiei recursive **`Fact`**.

- Este însă evident că în cazul calculului factorialului, recursivitatea poate fi înlocuită printr-o simplă **iteraţie** [5.2.2.b].

---

```
/*Calculul factorialului - implementare iterativă*/

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

---

- În figura 5.2.2.d. apare **reprezentarea grafică** a **profilului performanţei** algoritmului de **calcul al factorialului** în variantă **recursivă** respectiv **iterativă**.

  - Sunt de fapt reprezentaţi în manieră comparativă, **timpii de execuţie** pe un acelaşi sistem de calcul, ai celor doi algoritmi funcţie de valoarea lui `n`.

  - După cum se observă, deşi **ambii algoritmi** sunt **liniari** în raport cu `n`, adică au performanţa 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]

$T_r(n)=0.0056*n+0.017$

---

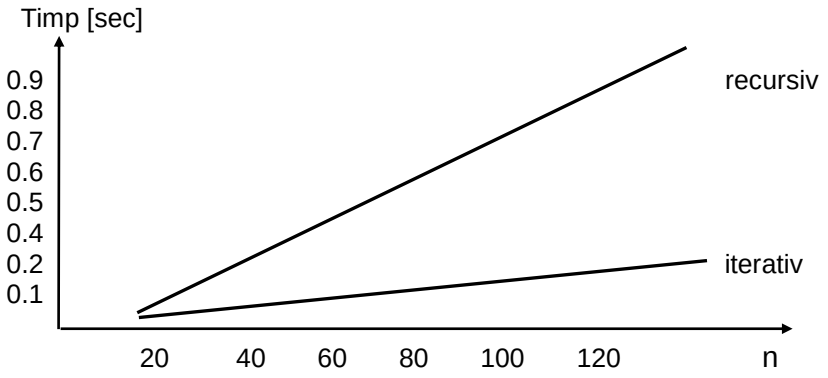



**Fig.5.2.2.d.** Profilul algoritmului factorial (variantele recursivă şi iterativă)

### 5.2.3. Calculul numerelor lui Fibonacci

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

---

$Fib_{n+1}=Fib_n+Fib_{n-1}$ pentru n>0 [5.2.3.a]

$Fib_1=1$; $Fib_0=0$

---

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

---

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

int fib(int n)                                    /*[5.2.3.b]*/
  {
    int fib_result;
    if(n==0)  fib_result=0;
      else
        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 situaţie conduce la o manieră ineficientă de calcul, deoarece numărul de apeluri creşte **exponenţial** cu `n`.
- În figura 5.2.3.a. apare reprezentarea arborelui de apeluri pentru `n=5`.

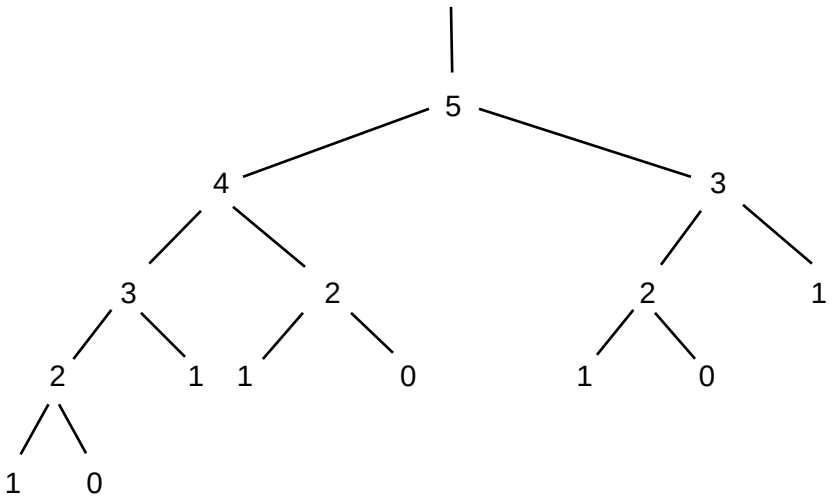



**Fig.5.2.3.a.** Arborele de apeluri al procedurii `Fib` pentru `n=5`

- 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.
- Ineficienţa acestei implementări creşte odată cu creşterea 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 $x=Fib_i$ şi $y=Fib_{i-1}$ [5.2.3.c].

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

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

- Variabila auxiliară `z` poate fi evitată, utilizând atribuiri de forma `x=x+y` şi `y=x-y`.

### 5.2.4. Eliminarea recursivităţii

- **Recursivitatea** reprezintă o **facilitate** excelentă de programare care contribuie la exprimarea simplă, concisă şi elegantă a algoritmilor de natură recursivă.
  - Cu toate acestea, ori de câte ori problemele eficienţei şi performanţei se pun cu preponderenţă, se recomandă **evitarea utilizării recursivităţii.**
  - De asemenea se recomandă evitarea recursivităţii ori de câte ori stă la dispoziţie o **rezolvare evidentă** bazată pe **iteraţie**.
- De fapt, implementarea recursivităţii pe echipamente nerecursive dovedeşte 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 **sfârşitul** ei, drept ultimă instrucţiune a procedurii ("**tail recursion**").
- În această situaţie, **recursivitatea** poate fi înlocuită cu o **buclă** simplă de **iteraţie**.
- Acest lucru este posibil, deoarece în acest caz, **revenirea** dintr-un apel încuibat presupune şi **terminarea** instanţei respective a procedurii, motiv pentru care contextul apelului **nu** mai trebuie restaurat.
- Astfel dacă o procedură `P(x)` conţine ca şi **ultim pas** al său un apel la ea însăşi de forma `P(y)`:

- Acest apel poate fi înlocuit cu o instrucţiune de atribuire `x=y`, urmată de reluarea (salt la începutul) codului lui `P`.
- `y` poate fi şi o expresie.
- `x` trebuie să fie în mod obligatoriu o variabilă transmisibilă prin **valoare**, astfel încât valoarea ei să fie memorată într-o locaţie specifică apelului.
- `x` poate fi transmis prin **referinţă** dacă `y` este chiar `x`.
- Dacă `P` are mai mulţi parametri, ei pot fi trataţi fiecare în parte ca `x` şi `y`.

- Această modificare este valabilă deoarece reluarea execuţiei lui `P`, cu noua valoare a lui `x`, are exact acelaşi efect ca şi când s-ar apela `P(y)` şi 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) ≡ P  [Si,IF c THEN P(y)]                    [5.2.4.a]
-------------------------------------------------------------
 P(x) ≡ P  [Si,IF c THEN [x=y; reluare P]]        [5.2.4.b]
-------------------------------------------------------------
 P(n) ≡ P  [Si,IF n>0 THEN P(n-1)]                [5.2.4.c]
-------------------------------------------------------------
 P(n) ≡ P  [Si,IF n>0 THEN [n=n-1; reluare P]]    [5.2.4.d]
-------------------------------------------------------------
```

  - Exemple concludente în acest sens sunt implementarea iterativă a **calculului factorialului** şi a **calculului şirului numerelor lui Fibonacci** prezentate anterior.

- (2) Cazul în care **apelul** sau **apelurile recursive** se realizează din **interiorul procedurii** [5.2.4.e].

```
-------------------------------------------------------------
 P ≡ P [Si, IF c THEN P, Sj]                      [5.2.4.e]
-------------------------------------------------------------
```

- Varianta iterativă a acestei situaţii presupune tratarea explicită de către programator a **stivei apelurilor**.
- Fiecare **apel** necesită **salvarea** în **stiva** gestionată de către utilizator a **contextului instanţei de apel**.
- Acest **context** trebuie **restaurat** de către utilizator din aceeaşi stivă la **terminarea instanţei** şi revenirea în instanţa anterioară de apel.
- Activitatea de conversie în acest caz are un **înalt grad de specificitate** funcţie de algoritmul recursiv care se doreşte a fi convertit, este în general mai dificilă, mai laborioasă şi îngreunează înţelegerea algoritmului.
  - Un exemplu în acest sens îl reprezintă **partiţionarea 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 algoritmii a căror **natură** este **recursivă** este indicat a fi formulaţi ca şi **proceduri recursive**, lucru pus în evidenţă de algoritmii prezentaţi în cadrul acestui capitol şi în capitolele următoare.

## 5.3. Exemple de algoritmi recursivi

### 5.3.1. Algoritmi care implementează definiţii recursive

- În continuare se prezintă două exemple de algoritmi care implementează definiţii 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 defineşte 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 înlocuieşte unul din numere cu restul împărţirii sale cu celălalt.
- Pornind de la această definiţie recursivă se poate concepe un subprogram simplu de tip funcţie pentru calculul c.m.m.d.c. a două numere întregi [5.3.1.1.a].

```
-------------------------------------------------------------
/*Implementarea algoritmului lui Euclid*/

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

- În figura 5.3.1.1.a apare urma execuţiei acestui algoritm pentru valorile `m=18` şi `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 execuţiei algoritmului lui Euclid

### 5.3.1.2. Algoritm recursiv pentru transformarea expresiilor aritmetice specificate în format infix în format postfix (notaţie 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 definiţii constă în faptul că definiţia lui **`factor`** include din nou definiţia lui **`expresie`**.

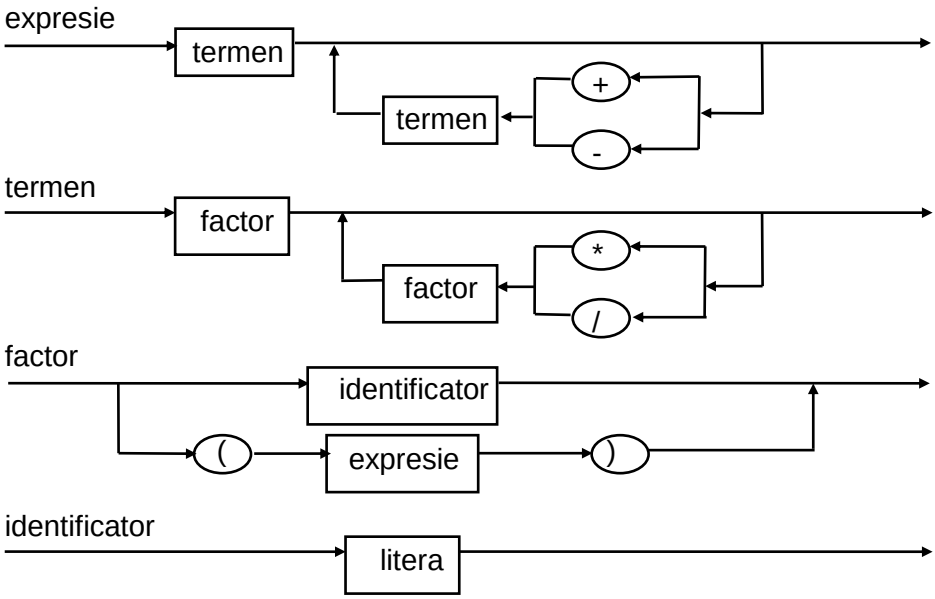



**Fig.5.3.1.2.a.** Definirea recursivă a unei expresii aritmetice

- Se pune **problema** ca pornind de la o **expresie aritmetică corectă** furnizată în format normal (**infix**) să se obţină forma poloneză (**postfix**) a acesteia.
- Se precizează că **notaţia poloneză** presupune reprezentarea unei operaţii aritmetice simple în forma *"operand operand operator"* şi nu în forma *"operand operator operand"* presupusă de notaţia infix.
    - Spre exemplu `a+b` devine `ab+` iar `a+b*(c-d)` devine `abcd-*+` .
- **Avantajul** notaţiei poloneze este acela că indică **ordinea corectă** a execuţiei operaţiilor **fără** a utiliza paranteze, iar expresia poate fi evaluată printr-o simplă **baleere secvenţială** de la stânga la dreapta.
- **Problema** care trebuie rezolvată este aceea de a **depista** cei **doi operanzi** (care pot fi complecşi), de a-i înregistra şi 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 `oa` respectiv `om`.

- Transformarea cerută se poate realiza construind câte o **procedură** individuală de conversie pentru fiecare construcţie sintactică (**expresie, termen, factor**).
- Deoarece fiecare dintre aceste construcţii sintactice este definită recursiv, procedurile corespunzătoare trebuiesc de asemenea activate recursiv.
- Algoritmul de transformare apare în secvenţa [5.3.1.2.a].

---

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

typedef char* linie;                        /*[5.3.1.2.a]*/
linie infix; /*expresie format infix*/
linie post;  /*espresie format postix*/
char c;
int i;       /*contor expresie curentă*/
int j;       /*contor expresie postfix*/

void expresie();
static void termen();

void citeste_car(linie ex, char* ch)
  /*furnizează caracterul următor al expresiei infix,
    eliminând blancurile*/
  {
    do {
      *ch=ex[i]; i++;
    } while (!(*ch!=' '));
  } /*citeste_car*/

void scrie_car(linie* ex, char ch)
  /*depune caracterul ch în zona precizată*/
  {
    (*ex)[j]=ch; j=j+1;
  } /*scrie_car*/

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 om;       /*variabilă operator multiplicativ*/
```

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

void expresie()
  {
    char oa;       /*variabilă operator aditiv*/
    termen();
    while ((c =='+')||(c =='-'))
      {
        oa=c;
        citeste_car(infix,&c);
        termen();
        scrie_car(&post,oa);
      } /*while*/
  } /*expresie*/
-----------------------------------------------------------------
```

- În cadrul **algoritmului de conversie** se definesc:
  - Procedurile **`expresie`**, **`termen`** şi **`factor`** conform diagramelor din fig. 5.3.1.2.a.
  - Variabile locale `oa` şi `om` utilizate pentru memorarea operatorilor **aditivi** respectiv **multiplicativi**.
  - Procedura **`citeste_car`** care furnizează caracterul următor din textul de analizat eliminând eventualele blancuri.
  - Variabila `ch` care în fiecare moment memorează caracterul furnizat de procedura **`citeste_car`**.
  - Se consideră că expresia în format **infix** ce urmează a fi convertită este depusă ca şir de caractere în tabloul `infix` parcurs cu ajutorul indicelui `i`.
  - Forma **postfix** a expresiei se asamblează în tabloul `post` utilizând în acest scop indicele `j`.
  - Indicii `i` şi `j` trebuiesc iniţializaţi în programul principal care apelează procedura **`expresie`**.
  - Pentru funcţionarea corectă a procedurii de conversie, **expresia de procesat** trebuie precizată între **paranteze**.

## 5.3.2. Algoritmi de divizare

### 5.3.2.1. Tehnica divizării (”Devide and Conquer”)

- Una dintre **metodele fundamentale de proiectare a algoritmilor** se bazează pe **tehnica divizării** ("**devide 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 şi din soluţiile cărora se poate **asambla** simplu soluţia problemei iniţiale.
  - (2) Se repetă **recursiv** pasul (1) până când subproblemele devin banale iar soluţiile lor evidente.
- O aplicaţie **tipică** a **tehnicii divizării** se bazează pe **modelul recursiv** prezentat în [5.3.2.1.a].

```
-------------------------------------------------------------
/*Tehnica divizării - soluţia recursivă - pseudocod */

procedure rezolvă(tip_problema x);        /*[5.3.2.1.a]*/

    daca(x este divizibil în subprobleme)
        |*divide pe x în subprobleme: x1,x2,..,xk;
        |rezolvă(x1);
        |rezolvă(x2);
        |    ...
        |rezolvă(xk);
        |*combină cele k soluţii parţiale într-o soluţie
        |   pentru x
        |□ /*daca*/
      altfel
         *rezolvă pe x direct;
 /*rezolva*/
-------------------------------------------------------------
```

- Dacă **recombinarea** soluţiilor parţiale este substanţial mai simplă decât rezolvarea întregii probleme, această tehnică conduce la proiectarea unor algoritmi într-adevăr eficienţi.
- 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 execuţie al rezolvării problemei de dimensiune `n` este `T(n)`.
- În condiţiile în care prin divizări succesive problema de rezolvat devine suficient de redusă ca dimensiune, se poate considera că pentru $n \le c$ ($c$ constant), determinarea soluţiei necesită un **timp de execuţie constant**, adică $\Theta(1)$.
- Se notează cu `D(n)` timpul necesar **divizării** problemei în subprobleme şi cu `C(n)` timpul necesar **combinării** soluţiilor parţiale.
- Dacă problema iniţială se divide în `k` **subprobleme**, fiecare dintre ele de dimensiune `1/b` din dimensiunea problemei originale, se obţine următoarea formulă recurentă [5.3.2.2.a].

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

### 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 soluţiilor** parţiale **nu** este necesară deoarece scopul este atins prin modificările care se realizează chiar de către rezolvările parţiale.

### 5.3.2.4. Algoritm pentru determinarea extremelor valorilor componentelor unui tablou liniar. Soluţia iterativă şi soluţia recursivă

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

```
/*Algoritm pentru deteterminarea extremelor unui vector - soluţia iterativă*/

#define n 100 /*dimensiunea tabloului*/  /*[5.3.2.4.a]*/
typedef int tip_tablou[n];

void domeniu_it(tip_tablou const a, int i,int j, int* mic,
int* mare)
/*atribuie parametrilor mic şi mare elementul minim
respectiv maxim al tabloului a din domeniul delimitat de
indicii i şi j respectiv (a[i]..a[j])*/
  {
     int k;
     *mic=a[i]; *mare=a[i];
     for(k=i+1;k<=j;k++)
      {
        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 uşor de văzut că **costul** procedurii **`DomeniuIt`** în termenii numărului de comparaţii dintre elementele tabloului este `2*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** ţine 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 condiţia de iniţializare şi reciproc.
  - Astfel algoritmul risipeşte un efort considerabil examinând fiecare element de două ori. Această risipă se poate evita.
- **O soluţia recursivă** bazată pe **tehnica divizării** ar putea fi următoarea [AH85]:
  - Se împarte tabloul în două părţi.
  - Se determină prin apelul recursiv al procedurii, valorile **minime** şi **maxime** ale fiecăruia dintre cele două tablouri.
  - Se compară valorile **minime** şi **maxime** determinate pentru subtablouri, stabilindu-se minimul şi maximul absolut.
  - În continuare se procedează în aceeaşi manieră reducând de fiecare dată la jumătate dimensiunea subtablourilor.
  - Pentru dimensiuni 1 sau 2 ale subtablourilor soluţia este imediată.
- O ilustrare a acestei tehnici apare în procedura recursivă **`Domeniu`** [5.3.2.4.b].

```
-------------------------------------------------------------
/*Algoritm pentru deteterminarea extremelor unui vector -
soluţia recursivă bazată pe tehnica divizării*/

#define n 100                                  /*[5.3.2.4.b]*/
typedef int tip_tablou[n];
tip_tablou a;

void domeniu(tip_tablou const a, int i,int j,int* mic,int*
mare)
/*atribuie parametrilor mic şi mare elementul minim
respectiv maxim al tabloului a din domeniul delimitat de
indicii i şi j respectiv (a[i]..a[j])*/
  {
    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
              {
                *mic=a[j-1]; *mare=a[i-1];
              }
        } /*if*/
```

```
      else        /*tablou de dimensiune mai mare ca 2*/
        {
          /*divide problema în două subprobleme*/
          mijloc=(i+j)/2;
          /*rezolvă subproblema 1*/
          domeniu(a,i,mijloc,&mic1,&mare1);
          /*rezolvă subproblema 2*/
          domeniu(a,mijloc+1,j,&mic2,&mare2);
          /*combină soluţiile parţiale*/
          if(mare1>mare2)
                *mare=mare1;
             else
                *mare=mare2;
          if(mic1<mic2)
                *mic=mic1;
             else
                *mic=mic2;
        } /*else*/
    } /*domeniu*/
```

---

- În figura 5.3.2.4.a. apare reprezentarea grafică a modului de lucru al procedurii Domeniu.

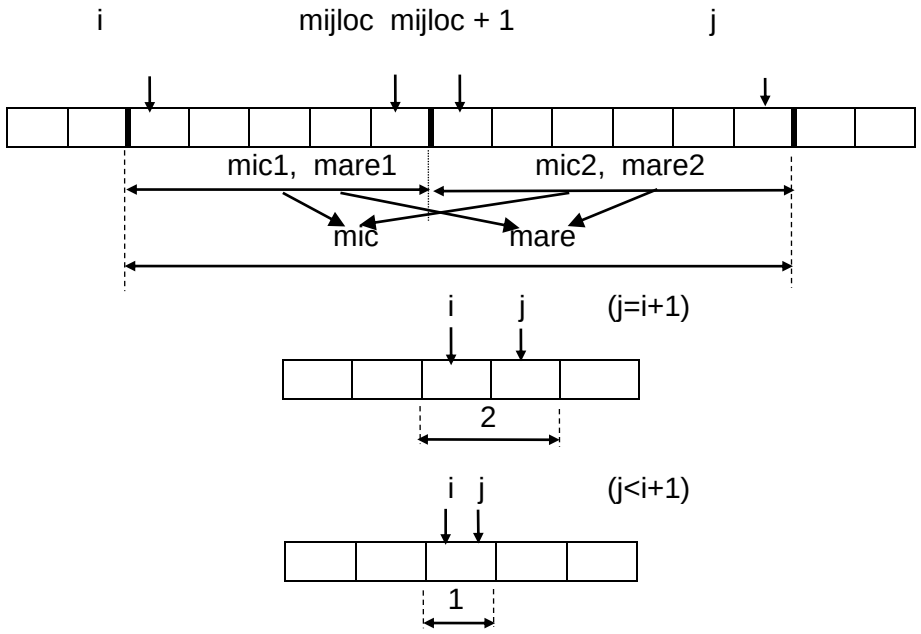



**Fig.5.3.2.4.a.** Funcţionarea de principiu a procedurii `Domeniu`.

- Se constată **analogia** evidentă cu structura de principiu prezentată în secvenţa [5.3.2.1.a].

- **Urma execuţiei** procedurii pentru `n=10` apare în figura 5.3.2.4.b.

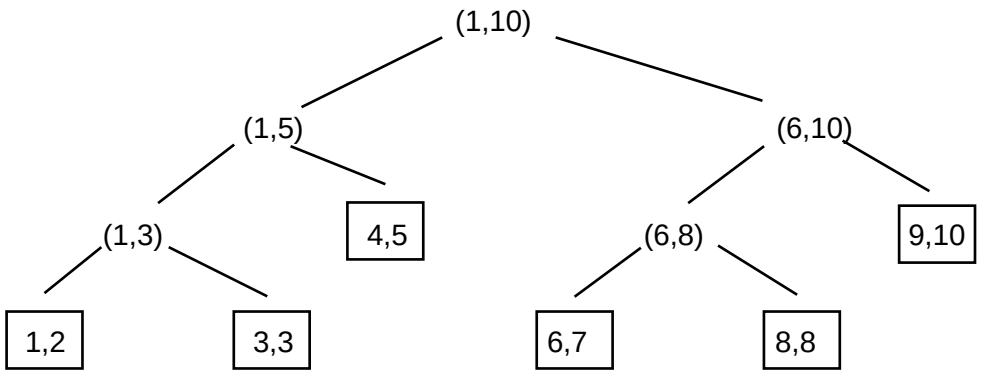



**Fig.5.3.2.4.b.** Arbore de apeluri al procedurii **`Domeniu`** pentru `n=10`

- În figura 5.3.2.4.b apar precizate limitele domeniilor pentru fiecare dintre apelurile procedurii.
- În chenar sunt precizate domeniile de lungime 1 sau 2 care presupun comparaţii 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 posibilităţilor** evidenţiate 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`),
- Totuşi, autorii demonstrează că din punctul de vedere al costului calculat relativ la **numărul de comparaţii** ale elementelor tabloului, el este mai eficient decât algoritmul iterativ.
- Pornind de la observaţiile:
    - (1) Procedura include comparaţii 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 comparaţii directe) egală cu `(3/2)n-2` unde `n` este dimensiunea tabloului analizat.
- Analiza cantitativă a algoritmului **`domeniu`** se poate realiza prin particularizarea relaţiei [5.3.2.2.a].

- Se observă imediat că `k=2, b=2,` iar `D(n)` şi `C(n)` sunt ambele $\Theta(1)$.

- În consecinţă rezultă ecuaţia [5.3.2.4.c] a cărei soluţie apare în [5.3.2.4.d].

---

$$T(n) = \begin{cases} \Theta(1) & daca\ n \le 2 \\ 2 \cdot T(n/2) + 2 & daca\ n > 2 \end{cases} \qquad [5.3.2.4.c]$$

---

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

---

### 5.3.3. Algoritmi de reducere

- O altă categorie de algoritmi care se pretează pentru o **abordare recursivă** o reprezintă **algoritmii de reducere**.
    - Aceşti 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 acceeaşi manieră **recursivă** şi se **asamblează** soluţia integrală.
- În această categorie pot fi încadraţi algoritmul pentru **calculul factorialului** şi algoritmul pentru **rezolvarea problemei Turnurilor din Hanoi**.
- Se atrage atenţia că spre deosebire de **algoritmii cu revenire** (&5.4), în acest caz **nu** se pune problema **revenirii** în caz de nereuşită, 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 secvenţă 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 obţine permutările de `n` elemente este suficient să se **fixeze** pe rând câte un element şi 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 **`permuta`**`(k)` care realizează permutarea a `k` numere.
- Schiţa de principiu a acestui algoritm apare în secvenţa [5.3.3.1.a.].

---

```
/*Determinarea permutărilor a k numere - schiţa de principiu
a algoritmului recursiv - varianta pseudocod*/

Procedura permuta(int k)                    /*[5.3.3.1.a]*/
 /*numerele se consideră memorate în tabloul a[0..k-1]*/
   | dacă k=0 atunci
   |     *afişează tabloul /*s-a finalizat o permutare*/
   |   altfel
   |     pentru i=0 la k-1 execută
   |       |*interschimbă pe a[i] cu a[k];
   |       |permuta(k-1);  /*apel recursiv*/
   |       |*interschimbă pe a[i] cu a[k] /*refacere stare*/
   |       | /*altfel*/
   | /*daca*/
/*permuta*/
```

---

- Numerele se presupun memorate în tabloul `a[i](i=0,1,...,n-1)`.
- Dacă `k` ≠ `0`, atunci fiecare dintre elementele tabloului situate pe poziţii **inferioare** lui `k` sunt aduse (fixate) pe poziţia `k` prin **interschimbarea** poziţiilor `i` şi `k` ale tabloului `a` în cadrul unei bucle repetitive pentru `i=0,1,...,k-1`.
- Iniţial `k` ia valoarea `n-1` pentru permutarea a `n` numere.
- Pentru fiecare schimbare (fixare) se **apelează** recursiv rutina cu parametrul `k-1`.
  - După revenire, se **reface** starea iniţială reluând în sens invers interschimbarea poziţiilor `i` şi `k`.
  - Acest lucru este necesar, deoarece fixarea elementului următor presupune **refacerea** stării iniţiale în vederea parcurgerii în mod ordonat, pentru fiecare element în parte a **tuturor** posibilităţilor.
- În momentul în care `k` = 0, se consideră **terminat** un apel şi se afişează tabloul `a` care conţine o permutare.
  - Aceasta este **condiţia** care limitează adâncimea apelurilor recursive determinând **revenirea** în apelurile anterioare.
- În secvenţele următoare apare implementarea algoritmului de determinare a permutărilor a `n` numere date [5.3.3.1.b.].

---

```
/*Exemplu de implementare a algoritmului pentru determinarea
permutărilor a  k numere naturale. Numerele se consideră
memorate în tabloul a[0..k-1]*/

int a[numar_elemente];  /*tablou elemente de permutat*/

void permuta(int k)                          /*[5.3.3.b]*/
  {
      int i,x;
      if (k==0)
         *afiseaza;
      else
        {
```

```
            for(i=0;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 obţinute prin fixarea primului element.
    - Pentru celelalte elemente, din lipsă de spaţiu, sunt doar sugeraţi subarborii corespunzători.

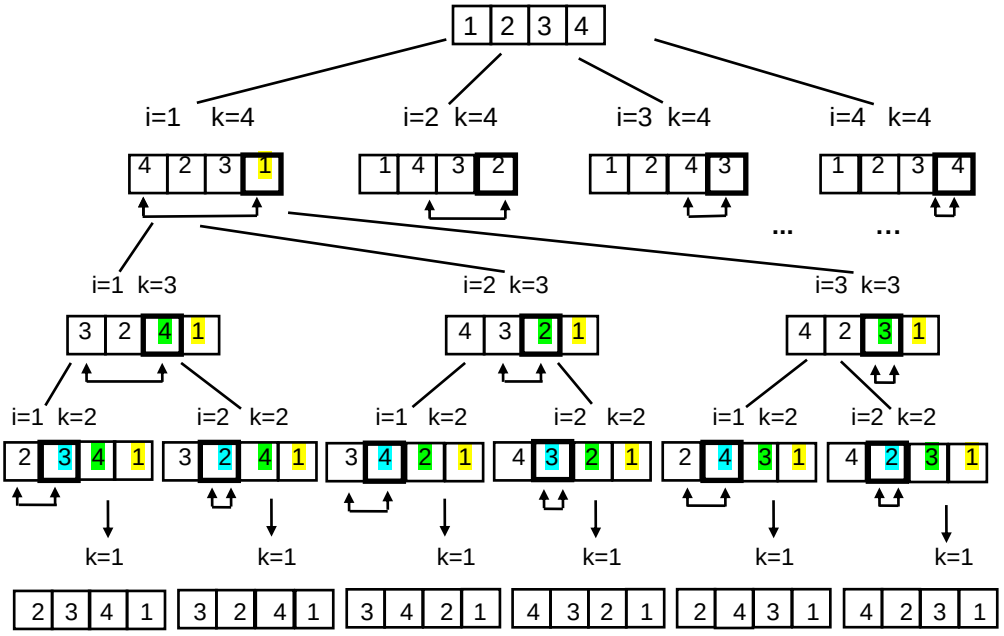



**Fig.5.3.3.1.a.** Arborele de apeluri pentru **`Permuta`**`(4)`

- Structura **arborelui apelurilor** este mai **complicată** datorită faptului că **apelurile** recursive se realizează dintr-o buclă **`for`** cu o limită `(k)` care **descreşte** odată cu creşterea nivelului arborelui.
    - **Înălţimea** arborelui de apeluri este egală cu `n`.

## 5.3.4. Algoritmi recursivi pentru determinarea tuturor soluţiilor unor probleme

- Algoritmii recursivi au **proprietatea** de a putea evidenţia în mod ordonat **toate posibilităţile** referitoare la o situaţie dată.
- Se prezintă în acest sens două exemple de astfel de algoritmi:
    - Primul exemplu reliefează proprietatea de a evidenţia în mod ordonat **toate posibilităţile** referitoare la o situaţie dată.

- Cel de-al doilea exemplu **selectează** din mulţimea tuturor posibilităţilor evidenţiate, **pe cele care prezintă interes**.

### 5.3.4.1. Algoritm pentru evidenţierea tuturor posibilităţilor de tăiere a unui fir de lungime dată

- Algoritmul ce urmează evidenţiază toate posibilităţile de secţionare a unui fir de lungime întreagă dată `(n)` în părţi de lungime 1 sau 2.
- Acest algoritm se bazează pe următoarea **tehnică** de lucru:
  - Pentru lungimea `n>1` există două posibilităţi:
    - Se **taie** o parte de **lungime 1** şi restul lungimii `(n-1)` se secţionează în **toate modurile posibile**.
    - Se **taie** o parte de **lungime 2** şi restul lungimii `(n-2)` se secţionează 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.
- Schiţa de principiu a acestui algoritm apare în secvenţa [5.3.4.1.a].

```
-------------------------------------------------------------
/*Schiţa de principiu a algoritmului de tăiere a firului*/

Procedura Taie(lungimeFir);                  /*[5.3.4.1.a]*/
  dacă(lungimeFir>1)
     |*se taie o bucată de lungime 1;
     |Taie(lungimeFir-1);
     |*se taie o bucată de lungime 2;
     |Taie(lungimeFir-2);
     |*se anulează tăietura;
     |  /*daca*/
    altfel
     |dacă(lungimeFir=1)
     |   *se taie bucata de lungime 1;
     |*afişare;
     |  /*altfel*/
/*Taie*/
-------------------------------------------------------------
```

- În secvenţa [5.3.4.1.b] se prezintă o variantă de implementare.
- Câteva detalii de implementare:
  - Procedura **`Taie`** implementează tehnica de lucru anterior prezentată cu precizarea că la atingerea dimensiunii `n=1` sau `n=0` se consideră procedura terminată şi se afişează secvenţa de tăieturi generată.
  - Tăieturile sunt reprezentate grafic utilizând caracterul '.' pentru segmentul de lungime 1 şi 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 de implementare a algoritmului de tăiere a
firului*/

int i,k,x;                                    /*[5.3.4.1.b]*/
char z[9]; /*tablou tăieturi*/

void taie(int lungime_fir)
  {
    if(lungime_fir>1)
        {
          k=k+1;
          z[k-1]='.';        /*taie o bucată de lungime 1*/
          taie(lungime_fir-1);
          z[k-1]='_';        /*taie o bucată de lungime 2*/
          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 uşor maniera de execuţie a procedurii (urma execuţiei) 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).
    -

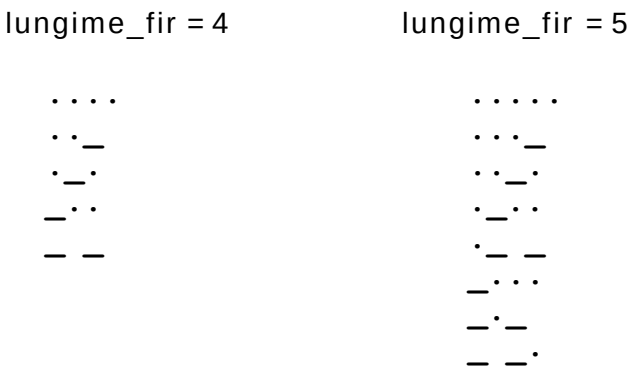



**Fig.5.3.4.1.a.** Execuţia procedurii `taie` pentru `n=4` şi `n=5`

- În figura5.3.4.1.b se prezintă **arborele de apeluri** al procedurii **`taie`** pentru `n = 4` .

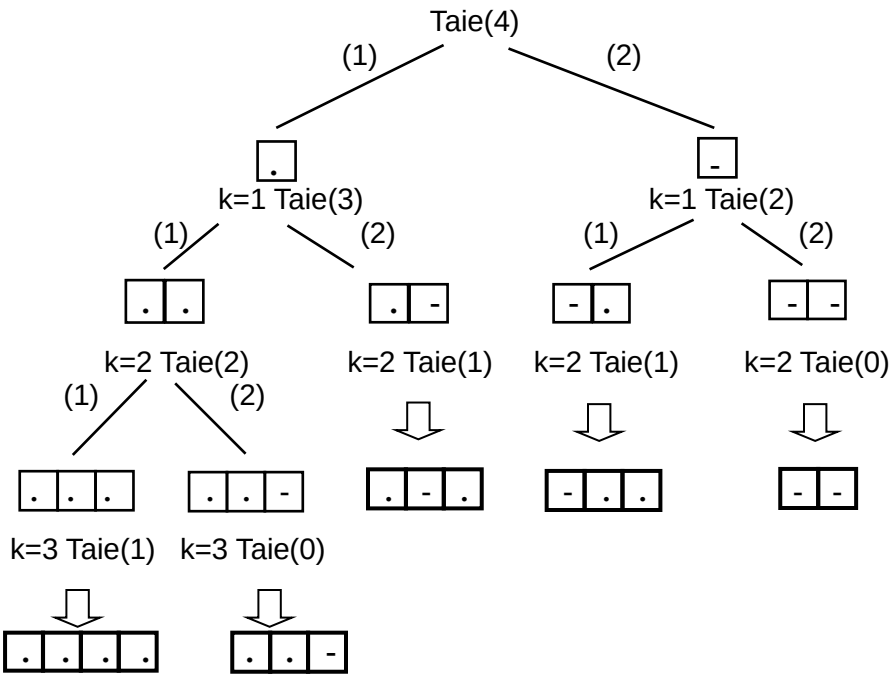



**Fig.5.3.4.1.b.** Arborele de apeluri al procedurii **`taie`**`(4)`

- Se observă că în fiecare nod al acestui arbore sunt precizate:
  - Succesiunea bucăţilor tăiate.
  - Valoarea curentă a lui `k`.
  - Apelul recursiv care se realizează.
  - În chenar îngroşat sunt încadrate secvenţele care se afişează.
- În baza modelului prezentat, acest algoritm poate fi extins cu uşurinţă pentru a determina posibilităţile de tăiere pentru pentru **orice număr precizat de tăieturi**, pentru **orice structură de lungimi**.

## 5.3.4.2. Algoritm pentru determinarea tuturor drumurilor de ieşire 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 reprezentaţi **pereţii labirintului**.
  - Cu ajutorul caracterului **`' '`** (spaţiu liber) sunt reprezentate **culoarele**.
  - Punctul de start este **centrul labirintului**.

- **Drumul de ieşire** se căută cu ajutorul unei **proceduri recursive** **`Cauta`**`(x,y)` unde `x` şi `y` sunt **coordonatele** locului curent în labirint şi în acelaşi timp **indici** ai tabloului care materializează labirintul.
- **Căutarea** se execută astfel:
- (1) Dacă valoarea locului curent este `' '` (spaţiu liber):
  - Se intră pe un **posibil** traseu de ieşire.
  - Se marchează locul cu caracterul `'#'`.
  - Dacă s-a ajuns la margine rezultă că s-a găsit un **drum de ieşire** din labirint şi se execută afişarea tabloului bidimensional (labirintul şi drumul găsit).
- (2) Dacă valoarea locului curent **nu** este `' '` (spaţiu liber):
  - Se apelează recursiv procedura **`Cauta`** pentru cele patru puncte din vecinătatea imediată a locului curent, după direcţiile axelor de coordonate (dreapta, sus, stânga, jos).
- Pentru fiecare căutare reuşită traseul apare marcat cu `'#'`.
- Marcarea asigură **nu** numai memorarea drumului de ieşire dar în acelaşi timp înlătură **reparcurgerea** unui drum deja ales.
  - Reluarea parcurgerii aceluiaşi drum poate conduce la un ciclu **infinit**.
- Este importantă sublinierea faptului că marcajul de drum se **şterge** de îndată ce s-a ajuns într-o **fundătură** sau dacă s-a găsit o **ieşire**, în ambele situaţii **revenindu-se** pe drumul parcurs până la proximul punct care permite selecţia unei **noi posibilităţi** de drum.
- **Ştergerea** se execută prin generarea unui caracter `' '` pe poziţia curentă înainte de părăsirea procedurii.
  - Aceasta corespunde practic înfăşurării "firului parcurgerii" conform metodei "firului Ariadnei".
- În secvenţa [5.3.4.2.a] este prezentată schiţa de principiu a procedurii **`Caută`** iar în [5.3.4.2.b] o variantă de implementare `C`.

---

```
/*Schiţa de principiu a algoritmului de căutare a drumului
de ieşire dintr-un labirint*/

Procedura Cauta(coordonate_loc x,y)       /*[5.3.4.2.a]*/
  dacă(locul este liber)
   |*marchează locul;
   |dacă(s-a ajuns la ieşire)
   |    *afişează drumul;
   |  altfel   /*continuă cautarea*/
   |   |Cauta(x+1,y);   /*caută dreapta*/
   |   |Cauta(x,y+1);   /*caută sus*/
   |   |Cauta(x-1,y);   /*caută stânga*/
   |   |Cauta(x,y-1);   /*caută jos*/
   |    /*altfel*/
   |*şterge marcajul;
    /*daca*/
```

```
/*Cauta*/
-------------------------------------------------------------
/*Variantă de implementare a algoritmului de căutare a
drumului de ieşire dintr-un labirint*/

void cauta(int x, int y)                    /*[5.3.4.2.b]*/
  {
    if(m[x][y]==' ')     /*locul este liber*/
      {
        m[x][y]='#';     /*marchează locul*/
        if(((x%n)==0)||((y%n)==0));  /*s-a ajuns la ieşire*/
            *afiseaza labirintul;  /*afişează drumul*/
          else  /*continuă cautarea*/
            {
              cauta(x+1,y);  /*caută dreapta*/
              cauta(x,y+1);  /*caută sus*/
              cauta(x-1,y);  /*caută stânga*/
              cauta(x,y-1);  /*caută jos*/
            }
        m[x][y]=' ';  /*ştergere marcajul*/
      }
  } /*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 ieşire din labirint şi se realizează afişarea.

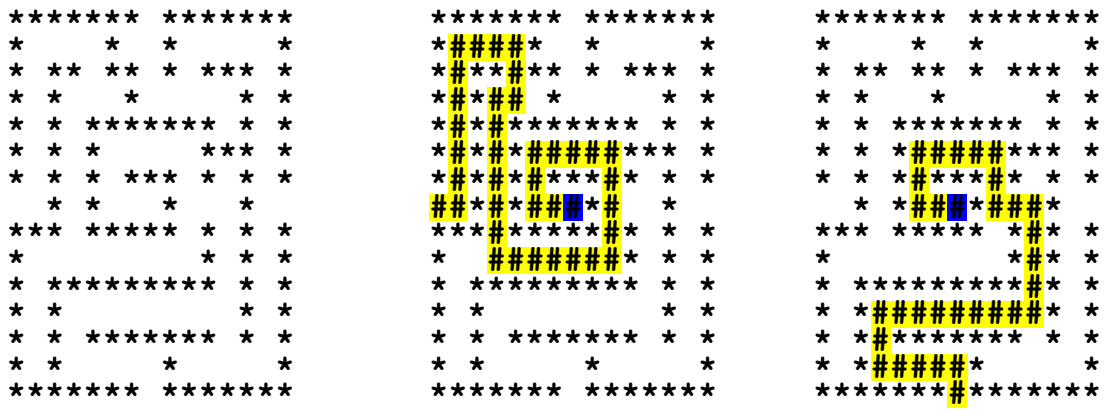


**Fig.5.3.4.2.a.** Exemplu de execuţie al procedurii `Cauta`

- În figura 5.3.4.2.a se prezintă un exemplu de execuţie al acestei proceduri.

## 5.3.5. Algoritmi pentru trasarea unor curbe recursive

- **Curbele recursive** reprezintă un alt domeniu de aplicaţie 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 esenţial la fundamentarea, consolidarea şi înţelegerea conceptului de recursivitate.
- Se face de asemenea precizarea că aceste curbe se pot include şi 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 $H_0$, $H_1$, $H_2$ şi $H_3$ care reprezintă curbele lui Hilbert de ordin 0, 1, 2 şi 3.
- Se observă că:
  - O curbă Hilbert $\mathbf{H_{i+1}}$ se obţine prin compunerea a patru instanţe de curbe $\mathbf{H_i}$.
  - Cele 4 instanţe au dimensiunea înjumătăţită.
  - Ele sunt rotite într-o manieră specifică.
  - Instanţele sunt legate între ele prin trei linii de legătură (prezentate îngroşat în desen).

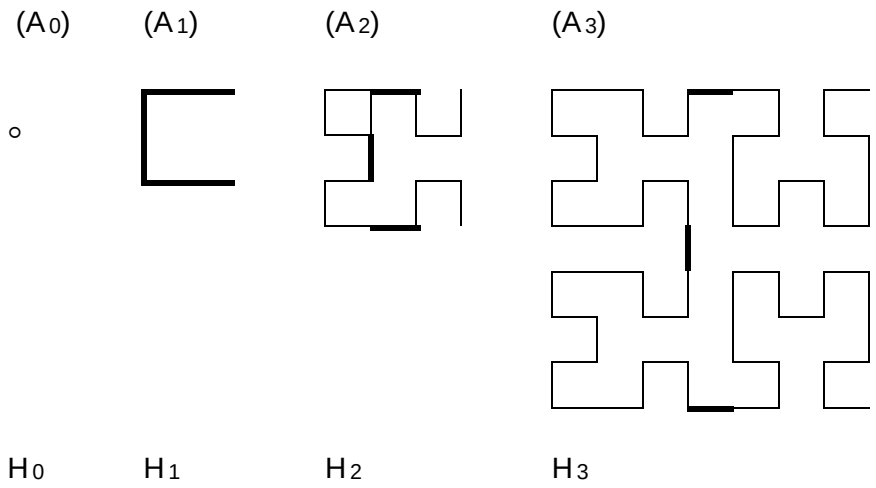



**Fig.5.3.5.1.a.** Curbele lui Hilbert de ordin 0, 1, 2 şi 3

  - Se mai observă că $H_1$ poate fi considerată ca şi constând din 4 instanţe al curbei vide $H_0$, unite prin aceleaşi 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 poziţionează cursorul grafic în punctul de coordonate `x,y;`

- **`LineTo`**`(x,y)` care trasează o linie dreaptă între poziţia curentă şi cea indicată prin coordonatele parametri `x,y`.

- În continuare ne propunem să concepem un program recursiv simplu pentru **generarea şi reprezentarea grafică a curbelor Hilbert** de diferite ordine.
  - Deoarece fiecare curbă $H_i$ constă din patru copii înjumătăţite $H_{i-1}$, este normal ca procedura care-l desenează pe $H_i$ să fie compusă din patru părţi, fiecare desenând câte o curbă $H_{i-1}$ rotită corespunzător şi la dimensiunea cerută.

- Prin generalizare se deduc cele patru **modele recursive** care sunt prezentate în figura 5.3.5.1.b.
  - Cele patru părţi sunt denumite A, B, C şi D, iar punctele lor de start sunt marcate cu cerculeţe.
  - Rutinele care trasează liniile sunt reprezentate prin săgeţi indicând direcţia de trasare.

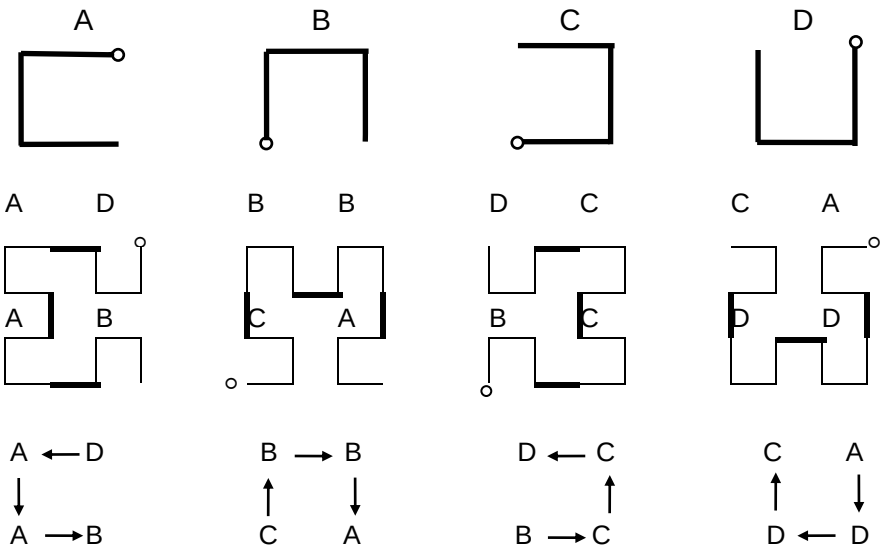



**Fig.5.3.5.1.b.** Modele recursive pentru curbele lui Hilbert

- 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                                    [5.3.5.1.a]
C:B→ C↑ C← D
D:A↓ D← D↑ C
-------------------------------------------------------------
```

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

```
-------------------------------------------------------------
/*Procedura pentru trasarea recursivă a modelului A al unei
curbe Hilbert - varianta pseudocod*/
```

```
                                                    /*[5.3.5.1.b]*/
int x,y; /*coordonatele cursorului grafic*/
int h;   /*lungimea liniei de conectare*/

PROCEDURE A(int i)
/*desenează modelul A:D←A↑A→B de ordinul i*/
    daca 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);
     |□ /*daca*/
 /*A*/
```

---

- Procedura este iniţializată în programul principal pentru fiecare curbă Hilbert care va fi supraimprimată.
    - Se consideră că pe acelaşi desen se suprapun curbe Hilbert de mai multe ordine succesive.
    - Ceea ce rezultă se aseamănă cu reprezentările **fractalilor**.
- Programul principal determină **punctul iniţial de start** şi **lungimea curenta a liniei** `h` pentru fiecare curbă generată.
- Variabila `h0` care precizează dimensiunea totală a paginii trebuie să aibă valoarea $2^k$ unde $k > n$, $n$ fiind ordinul maxim al curbelor care supraimprimă.
- Programul integral care trasează curbele Hilbert $H_1$, $H_2$, ..., $H_n$ apare în secvenţa [5.3.5.1.c] .

---

```
/* Program pentru trasarea recursivă a unor curbe Hilbert de
diferite ordine –varianta C */

/*deseneaza curbele hilbert de ordin 1,2,...,n*/
#include <graphics.h>                    /*[5.3.5.1.c]*/
#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:D←A↓A→B */
  {
    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*/

void b(int i) /*desenează modelul B:C↑B→B↓A */
  {
    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:B→C↑C←D */
  {
    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:A↓D←D↑C */
  {
    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 pricipal*/
  graphic_init();
  i=0;
  h=h0;    /*valoare inițială pentru lungimea liniei*/
  x0=h/2;  /*coordonate inițiale pentru punctul de pornire*/
  y01=x0;
  do {  /*trasează curba hilbert de ordin i*/
      i++;
      h=h / 2;
      x0=x0+(h/2); /*coordonatele punctului de start al
                     curbei curente*/
      y01=y01+(h/2);
      x=x0; y=y01;
      moveto(x,y);
      a(i);
  } while(!(i==n));
  getchar();
  graphic_close();
  return 0;
```

```
}
```

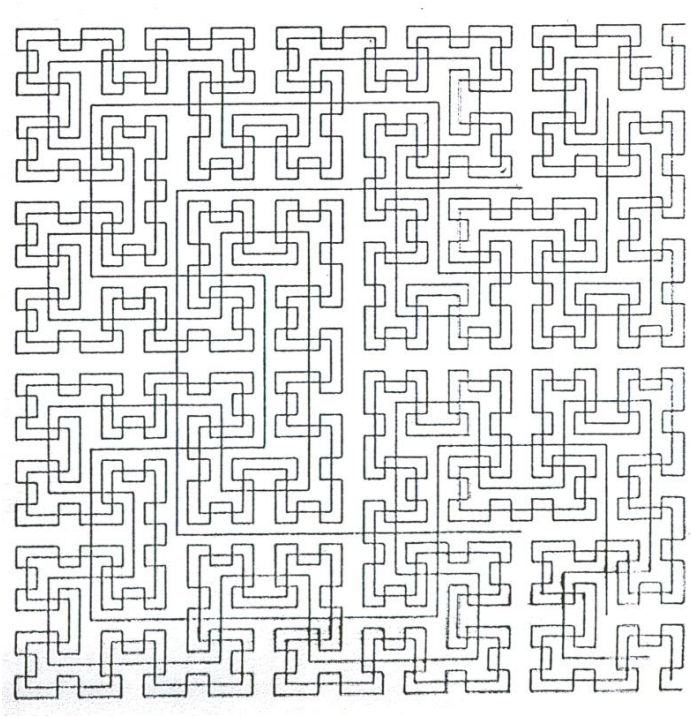


**Fig.5.3.5.1.c.** Curbele lui Hilbert $H_1$ – $H_5$

- În figura 5.3.5.1.c sunt reprezentate curbele lui Hilbert până la ordinul 5, rezultate din execuţia 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 ($S_1$ şi $S_2$).

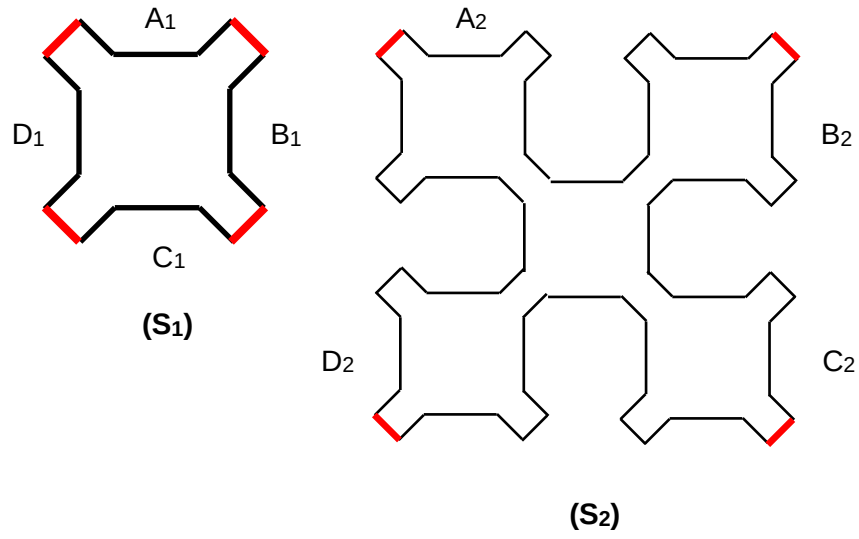



**Fig.5.3.5.2.a.** Modele recursive pentru curbele lui Sierpinski de ordin 1 şi 2

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

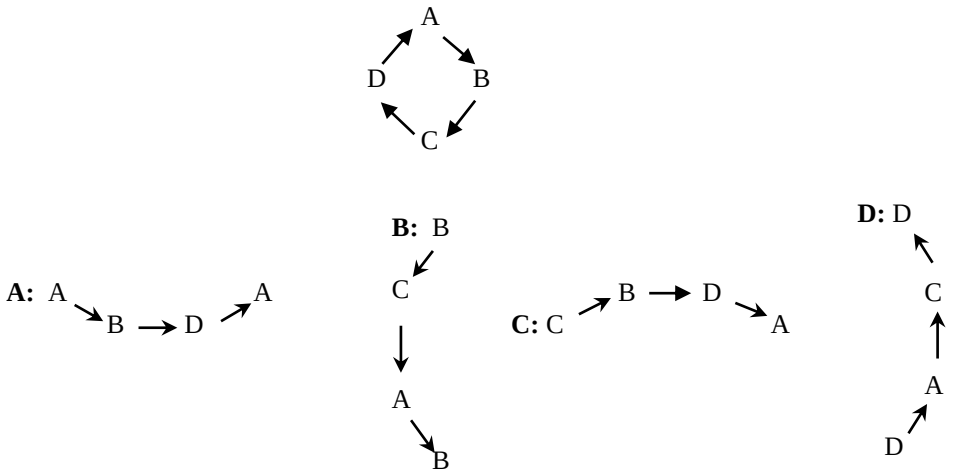



**Fig.5.3.5.2.b.** Modelele recursive ale curbelor lui Sierpinski

- Din aceste **modele recursive** pot fi deduse simplu **schemele recursive** care permit generarea curbelor.
- În secvenţa [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↘ B↙ C↖ D↗ [5.3.5.2.a]

---

A : A↘ B ⇒ D↗ A
B : B↙ C ⇓ A↘ B [5.3.5.2.b]
C : C↖ D ⇐ B↙ C
D : D↗ A ⇑ C↖ D

---

- Considerând aceleaşi facilităţi de reprezentare ca şi în cazul anterior pot fi redactate cu uşurinţă 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 şi procedurile **B,C** şi **D**.
- În secvenţa [5.3.5.2.c ] apare un model de implementare pentru procedura **A** .

---

**/*Exemplu de implementare a modelului A pentru o curba Sierpinski varianta pseudocod*/**

/*[5.3.5.2.c]*/

```
int x,y; /*coordonatele cursorului grafic*/
int h;   /*lungime linie de conectare*/

PROCEDURE A(i: integer);
/*deseneaza modelul   A:D↘ A⇒A ↗ B de ordinul i*/
    daca 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);
     |□ /*daca*/
 /*A*/
```

---

- **Programul principal** este conceput conform modelului [5.3.5.2.a].
- Sarcina **programului principal** este aceea de a iniţia generarea succesivă de curbe Sierpinski de ordine crescătoare şi de a fixa pentru fiecare, valoarea lui `h` şi valorile coordonatelor iniţiale conform cu dimensiunea zonei de afişare.
- Tot în sarcina programului principal cade şi trasarea celor patru linii care închid curbele trasate (modelul de bază).
- Programul integral apare în [5.3.5.2.d].
    - Eleganţa şi oportunitatea utilizării recursivităţii în această situaţie este evidentă.
    - Corectitudinea programului poate fi uşor dedusă din structura sa şi din forma modelelor.
    - Limitarea adâncimii recursivităţii s-a realizat cu ajutorul parametrului `i`, garantând în acest mod terminarea programului.
    - Utilizarea iteraţiei în aceste situaţii conduce la programe complicate şi greoaie.

---

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

```
#include <graphics.h>                    /*[5.3.5.2.d]*/

 /*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*/
     {
       ...
     }   /*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 execuţie a programului de generare şi reprezentare a curbelor lui Sierpinski pentru `n = 4`.

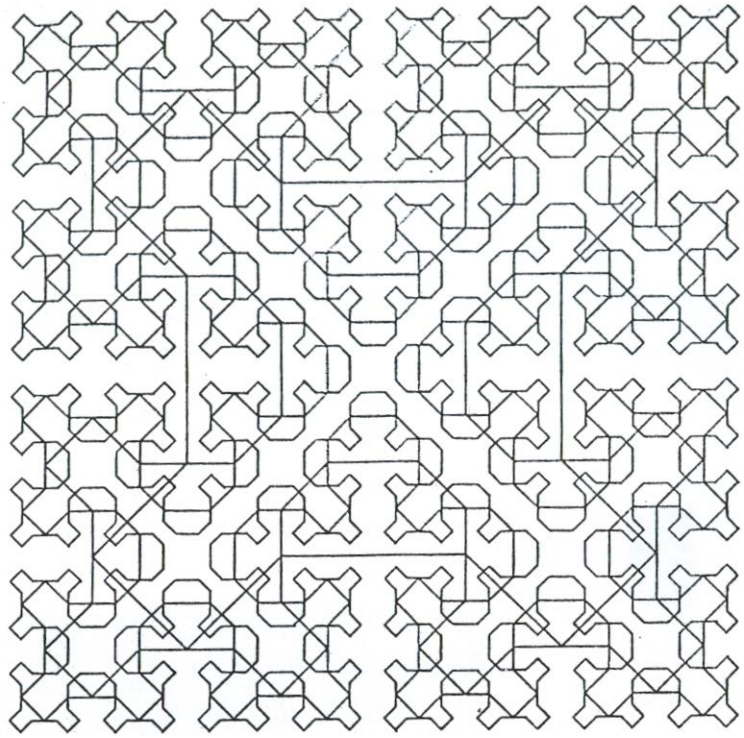


**Fig.5.3.5.2.c.** Curbele lui Sierpinski de ordinele $S_1$ – $S_4$

## 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 soluţiilor unor probleme specifice, care să **nu** se bazeze pe un **set fix de reguli de calcul**, ci pe **încercări repetate** şi **reveniri** în caz de nereuşită.
- Modalitatea comună de realizare a acestei tehnici constă în **descompunerea obiectivului** (taskului) în **obiective parţiale** (taskuri parţiale).
    - De regulă această descompunere este exprimată în mod natural în **termeni recursivi** şi constă în explorarea unui **număr finit** de subtaskuri.
    - În general, întregul proces poate fi privit ca un proces de **încercare** sau **căutare** care **construieşte** în mod gradat **soluţia** şi **parcurge** în acelaşi timp **un arbore de subprobleme.**
    - Obţinerea unor **soluţii parţiale** sau **finale** care **nu** satisfac, provoacă **revenirea recursivă** în cadrul procesului de căutare şi **reluarea** acestuia până la obţinerea **soluţiei 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 **exponenţial**, iar efortul de căutare creşte în aceeaşi măsură.
    - În mod obişnuit, arborii de căutare pot fi simplificaţi numai cu ajutorul **euristicilor**, simplificare care se reflectă de fapt în restrângerea volumului de calcul şi încadrarea sa în **limite acceptabile**.
- **Scopul** acestui paragraf:
    - **Nu** este acela de a discuta principiile şi 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 soluţionării acestora.

### 5.4.1. Turneul calului

- **Specificarea** problemei **turneului calului**:
    - Se consideră o tablă de şah (eşicher) cu $n^2$ câmpuri.
    - Un cal căruia îi este permis a se mişca conform regulilor şahului, este plasat în câmpul cu coordonatele iniţiale $x_0$ şi $y_0$.
    - 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 execuţia unei **mişcări următoare** sau se constată că **nu** mai este posibilă nici una şi atunci se **revine** în pasul anterior".
- Pentru început se va defini algoritmul care încearcă să execute **mişcarea următoare** a calului.

```
-------------------------------------------------------------
/*Schiţa algoritmului pentru realizarea mişcării următoare a
calului - varianta pseudocod*/

void încearcă_miscarea_următoare()          /*[5.4.1.a]*/

    *iniţializează lista mişcărilor;
   executa
     *selectează posibilitatea următoare din lista
        mişcărilor;
     daca(este acceptabilă)
      *înregistrează miscarea curentă;
      daca(tabela nu e plină)
           încearcă_mişcarea_următoare();
           daca(!mişcare reusită)
              *şterge înregistrarea curentă;
           □ /*daca*/
        altfel
          *mişcare reuşită;
      □ /*daca*/
   pana cand(!((mişcare reusită) || (nu mai există
                posibilităţi în lista mişcărilor)));
    □ /*executa*/
 /*încearcă_mişcarea_următoare*/
-------------------------------------------------------------
```

- În continuare se vor preciza **structurile de date** care vor fi utilizate.
- În primul rând **tabla de şah** va fi reprezentată prin matricea `t` [5.4.1.b].

```
-------------------------------------------------------------
/*Definirea structurilor de date: tabla de şah*/

enum {n=8};                                 /*[5.4.1.b]*/

typedef int tip_tabela[n][n];
tip_tabela t;
-------------------------------------------------------------
```

- După cum se observă, în câmpurile tabelei `t` se memorează valori întregi şi **nu** valori booleene care să indice ocuparea.

- Acest lucru se face cu scopul de a păstra **urma traseului parcurs** conform următoarei **convenţii** [5.4.1.c].

```
-------------------------------------------------------------
{Convenţia de marcare a traseului parcurs de cal pe tabela
de şah}

tip_tabela t;                               /*[5.4.1.c]*/
```

```
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 unde
                (1 ≤ i ≤ n²)*/
```

---

- În continuare se stabilesc **parametrii de apel** ai procedurii. Aceştia trebuie să precizeze:
  - (1) **Condiţiile de start** ale noii mişcări (parametri de intrare).
  - (2) Dacă mişcarea respectivă este **reuşită** sau **nu** (parametru de ieşire).
- Pentru prima cerinţă (1) sunt necesare:
  - **Coordonatele** `x,y` de la care va porni mişcarea.
  - **Valoarea** `i` care precizează **numărul mutării curente** (din raţiuni de înregistrare şi evidenţă).
- Pentru cea de-a doua cerinţă (2) se introduce **parametrul de ieşire** `q=`**`true`** desemnând mişcare reuşită respectiv `q=`**`false`** mişcare nereuşită, din punctul de vedere al acceptării mişcării.
- Problema care se pune în continuare este aceea a **rafinării** propoziţiilor precedate de caracterul "*****" în secvenţa 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` şi `v` pentru a preciza coordonatele unei **posibile destinaţii a mişcării** conform regulilor după care se efectuează saltul calului, atunci:
    - Propoziţia *****`este acceptabilă` poate fi exprimată ca şi o combinaţie logică a condiţiilor $1\le u\le n$ **şi** $1\le v\le n$ (adică noul câmp este pe tabelă) **şi** că el **nu** a fost vizitat anterior `(t[u,v]==0)`.
  - Aserţiunea *****`înregistrează mişcarea` devine `t[u,v]=i`.
  - Aserţiunea *****`şterge înregistrarea curentă`, se exprimă prin `t[u,v]=0`.
  - Se mai introduce **variabila booleană locală** `q1` utilizată ca şi parametru rezultat intermediar pentru apelul recursiv al procedurii. Variabila `q1` **substituie** de fapt condiţia *****`mişcare reuşită`.
- Se ajunge în definitiv la următoarea formulare a procedurii [5.4.1.d].

---

```
/*Rafinarea procedurii încearcă - pasul 1 de rafinare*/

typedef int tipindice;                          /*[5.4.1.d]*/
typedef unsigned int boolean;
#define true (1)
#define false (0)

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

```
    boolean q1;          /*variabilă surogat parametru q*/

    *iniţializează lista mişcărilor
    do
     {
       *fie u,v coordonatele mişcării următoare conform
         regulilor şahului;
       if((1<=u<=n)&&(1<=v<=n)&&(t[u][v]==0))
         {               /*mişcarea este acceptabilă*/
           t[u][v]=i;    /*înregistrează mişcarea*/
           if(i<n*n)     /*tabela nu este plină*/
              {
                incearca(i+1,u,v,&q1);/*mişcarea următoare*/
                if(!q1)  t[u][v]=0;   /*şterge mişcarea*/
              }
            else         /*tabela este plină*/
              q1=true;   /*mişcare reuşită*/
        }
    } while (!(q1||(*nu mai există posibilităţi în lista
                      mişcărilor)));
  *q=q1;                 /*restaurare variabila parametru q*/
 } /*incearca*/
```

---

- Relativ la această secvenţă 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` şi `y`.
  - (2) Se selectează **o nouă mişcare de coordonate** `u` şi `v` (de fapt următoarea mişcare din cele 8 posibile din lista de mişcări).
  - (3) Se încearcă **realizarea mişcării următoare** plecând de la poziţia `u,v`.
    - Dacă mişcarea **nu** este reuşită, respectiv s-au parcurs fără succes toate cele 8 posibilităţi ale listei de mişcări plecând de la `u` şi `v`, se **anulează mişcarea curentă** (`u,v`), ea fiind lipsită de perspectivă (bucla **`do-while`**).
- Privind din perspectiva evoluţiei căutării, fiecare dintre cele 8 mişcări este tratată în manieră similară:
  - Pornind de la fiecare dintre mişcări se merge atât de departe cât se poate.
  - În caz de nereuşită se încercă mişcarea următoare din lista de mişcări până la epuizarea tuturor posibilităţilor.
  - Dacă s-au epuizat toate posibilităţile, se **anulează** mişcarea curentă.
- După cum se observă, procedura se extinde de fapt peste **trei niveluri de căutare**, element care îi permite **revenirea** în caz de eşec 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 posibilităţi de mişcare).

- Are **înălţimea** $n^2$ (numărul de paşi necesari pentru soluţia finală), element care explică **complexitatea** procesului de determinare a soluţiei problemei.

- În pasul următor şi ultimul de rafinare mai rămân câteva puncte de specificat.
- Precizarea **saltului calului**.
  - Fiind dată o poziţie iniţială <x,y> există opt **posibilităţi** pentru generarea coordonatelor destinaţiei mişcării următoare <u,v>, care sunt numerotate de la 1 la 8 în fig.5.4.1.a.

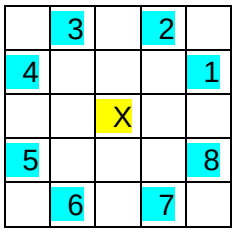



**Fig.5.4.1.a.** Mişcările posibile ale calului pe eşicher

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

```
-------------------------------------------------------------
                a[1]= 2; b[1]= 1;              /*[5.4.1.e]*/
                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;
-------------------------------------------------------------
```

- Detaliile de implementare apar în programul [5.4.1.e].
- Procedura recursivă este **iniţiată** printr-un apel cu coordonatele x0,y0 de la care porneşte 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 şi v sunt în domeniul 1..n.

```
-------------------------------------------------------------
/*Determinarea Turneului Calului - varianta finală*/

#include <limits.h>                          /*[5.4.1.f]*/
```

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

enum {n =5};

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

int i,j;              /*indici acces tabela de şah*/
boolean q;            /*variabilă succes mişcare*/
int a[8],b[8];        /*tabele diferenţe coordonate salt*/
int t[n][n];          /*tabela de şah*/

void incearca(int i, int x, int y, boolean* q)
    {
     int k;           /*indice în lista de mişcări*/
     int u,v;         /*coordonate mişcare următoare*/
     boolean q1;      /*variabilă surogat parametru q*/
     k=0;             /*iniţializare listă mişcări*/
     do {
       k=k+1;         /*selecţie mişcare următoare*/
       q1=false;
       /*calcul coordonate mişcare următoare*/
       u=x+a[k-1]; v=y+b[k-1];
       if ((0<=u<=n-1)&&(0<=v<=n-1)) /*mişcare pe tablă*/
          if (t[u-1][v-1]==0)        /*locaţie liberă*/
            {
              t[u-1][v-1]=i;    /*înregistrează mişcarea*/
              if (i<n*n)        /*tabela nu este plină*/
                  { /*încearcă mişcarea următoare*/
                    incearca(i+1,u,v,&q1);
                    if (!q1)          /*mişcarea nereuşită*/
                       t[u-1][v-1]=0; /*şterge mişcarea*/
                  }
                else              /*tabela este plină*/
                  q1=true;        /*mişcarea reuşită*/
            }
     } while (!(q1||(k==8))); /*mişcare reuşită sau epuizare
                                mişcări*/
      *q=q1;   /*restaurare variabilă parametru q*/
    } /*incearca*/

int main(int argc, const char* argv[])
  {     /*programul principal*/
    /*iniţializare tabele diferenţe coordonate salt*/
    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;
    /*iniţializare tabela de şah*/
```

```
    for(i=1;i<=n;i++)
      for(j=1;j<=n;j++)
         t[i-1][j-1]=0;
    /*iniţializare punct de start*/
    t[0][0]=1;
    /*determinare traseu turneu cal*/
    incearca(2,1,1,&q);
    if(q) /*determinare reuşită - afişare tabelă turneu*/
       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 execuţiei programului `TurneulCalului` pentru poziţiile iniţiale `(1,1),(3,3)` cu `n = 5` şi `(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 execuţie ale programului `TurneulCalului`

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

### 5.4.2. Probleme (n,m). Determinarea unei soluţii

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

```
-------------------------------------------------------------
/*Model pricipial de rezolvare a unei probleme de tip (n,m).
Determinarea unei soluţii*/

void incearca()                                  /*[5.4.2.a]*/
    *iniţializează selecţia posibilităţilor;
    executa
      *selectează posibilitatea următoare;
      daca(acceptabilă)
          *înregistreaz-o ca şi curentă;
          daca(*soluţie incompletă)
              incearca *pasul urmator;
              daca(*nu este reuşit)
                *şterge înregistrarea curentă;
              □ /*daca*/
            altfel
              *pas reusit(soluţie completă);
          □ /*daca*/
    pana cand ((*pas reuşit||(*nu mai sunt posibilităţi)
    □ /*executa*/
 /*incearca*/
-------------------------------------------------------------
/*Model pricipial de rezolvare a unei probleme de tip (n,m).
Determinarea unei soluţii - varianta pseudocod  C-like*/

void incearca()                                  /*[5.4.2.a]*/
  {
    *initializeaza selectia posibilitatilor;
    do
      *selecteaza posibilitatea urmatoare;
      if (*acceptabila)
        {
          *înregistreaz-o ca si curenta;
          if (*solutie incompleta)
              {
                incearca *pasul urmator;
                if (*nu este reusit)
                  *sterge înregistrarea curenta;
              }
```

```
            else
              *pas reusit (solutie completa);
        }/*if */
    while (!((*pas reusit)||(*nu mai sunt
                                posibilitati)))
  }/*incearca*/
```

---

- Acest **model principial** 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 posibilităţilor se realizează în bucla interioară **`do-while`** [5.4.2.b].

---

```
/*Rezolvarea unei probleme de tip (n,m). Determinarea unei
soluţii - Varianta 1 pseudocod C*/

void incearca1(Tip_Pas i)                    /*[5.4.2.b]*/
 /*parametrul de apel este numărul pasului curent*/
  {
    Tip_Posibilitate posibilitate;
    posibilitate=0;/*iniţializează selecţia
                     posibilităţilor*/
    do
      posibilitate=posibilitate+1; /*selecţie posibilitate
                                     următoare*/
      if(*acceptabilă)
        {
          *înregistreaz-o ca şi curentă;
          if(i<n) /*soluţie incompletă*/
              {
                incearca1(i+1); /*încearcă pasul următor*/
                if(*nereuşit)
                   *şterge înregistrarea curentă
              }
            else
              *soluţie completă (afişare)
        }
    while ((!*soluţie completă||(posibilitate!=m))
  } /*incearca1*/
```

---

- (2) În cea de-a doua **variantă**:
  - Procedura **`incearca2`** are drept parametru de apel **o posibilitate de selecţie**.
  - Construcţia soluţiei se realizează apelând în manieră recursivă procedura, succesiv, pentru fiecare posibilitate în parte [5.4.2.c].
- Din punctul de vedere al **finalităţii** cele două variante sunt **identice**, ele diferă doar ca formă.

---

```
/*Rezolvarea unei probleme de tip (n,m). Determinarea unei
soluţii - Varianta 2 pseudocod C*/

void incearca2(Tip_Posibilitate posibilitate) /*[5.4.2.c]*/
 /*parametrul de apel este o posibilitate de selecţie*/
  {
    if(*acceptabilă)
      {
        *înregistreaz-o ca şi curentă;
        if(*soluţie incompletă)
            {
              incearca2(posibilitate_1);
              incearca2(posibilitate_2);
                  ...
              incearca2(posibilitate_m);
              *şterge înregistrarea curentă;
            }
          else
            *soluţie completă (afişare);
      }
  } /*incearca2*/
```

---

- Se presupune că la fiecare pas numărul posibilităţilor de examinat este **fix** având valoarea `m` şi că procedura este apelată iniţial prin **`incearca2`**`(1)`.
- În continuare în cadrul acestui paragraf vor fi prezentate câteva **aplicaţii** 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 soluţie 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 şi acurateţe, atribute în care maşina de calcul excelează asupra omului chiar atunci când acesta este un geniu.
- **Specificarea problemei celor 8 regine**:
    - Pe o tablă de şah trebuiesc plasate 8 regine astfel încât nici una dintre ele să nu le ameninţe pe celelalte.
- Se observă imediat că aceasta este o problemă de tip `(n,m)`:
    - Există 8 regine care trebuiesc plasate pe eşicher, deci soluţia necesită 8 **paşi.**
    - Pentru fiecare din cele 8 regine există, după cum se va vedea, 8 **posibilităţi** de a fi aşezate pe tabla de şah.

- Pornind de la modelul [5.4.2.a] se obţine imediat următoarea **formulare primară** a **algoritmului** [5.4.3.a].

```
-------------------------------------------------------------
/*Rezolvarea Problemei celor 8 regine - schiţa de principiu
- pseudocod C*/

void incearca(regina i)                          /*[5.4.3.a]*/

    *iniţializează selecţia locului de plasare pentru
       a i-a regina
    do
      *selectează locul urmator;
      if(*loc sigur)
        *plasează regina i;
        if(i<8)
            incearca(i+1);
            if(*încercare nereuşită) *ia regina;
            □ /*if*/
          else
            *încercare reuşită (i=8);
        □ /*if*/
    while (!((*încercare reuşită)||(*nu mai există locuri)))
    □ /*do*/
/*incearca*/
-------------------------------------------------------------
```

- Sunt necesare câteva **precizări**.
    - Deoarece din regulile şahului se ştie că regina ameninţă **toate** câmpurile situate pe aceeaşi **coloană**, **rând** sau **diagonale** în raport cu câmpul pe care ea se află, **rezultă** că fiecare **coloană** a tablei de şah va putea conţine **o singură** regină.
    - Drept urmare alegerea poziţiei celei de-a `i`-a regine poate fi restrânsă **numai la coloana** `i`.
- În **consecinţă**:
    - Parametrul `i` din cadrul algoritmului devine **indexul coloanei** pe care va fi plasată regina `i`.
    - Procesul de selecţie 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).
    - Soluţionarea ei necesită 8 **paşi** (aşezarea celor 8 regine).
    - Fiecare regină se aşează într-una din cele 8 poziţii ale coloanei proprii (8 **posibilităţi**).
    - **Arborele de apeluri recursive** este de ordinul 8 şi are înălţimea 8.
- În continuare se impune alegerea modalităţii de **reprezentare a poziţiei** celor 8 regine pe tabla de şah.

- **Soluţia imediată** este aceea a reprezentării **tablei de şah** cu ajutorul unei **matrice** `t` de dimensiuni `[8,8]`.
    - O astfel de reprezentare conduce însă la operaţii greoaie şi complicate de determinare a câmpurilor disponibile.

- Pornind de la principiul că de fiecare dată trebuiesc utilizate reprezentările directe cele mai relevante şi mai eficiente ale informaţiei, în cazul de faţă **nu** se vor reprezenta poziţiile reginelor pe tabla de şah ci faptul că o regină a fost sau nu plasată pe o anumită **coloană,** pe un anumit **rând,** sau pe o anumită **diagonală**.
    - Ştiind 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*/
    int x[8];                                  /*[5.4.3.b]*/
    int y[8];
    boolean b[b2];
    boolean c[c2];
-------------------------------------------------------------
```

- Presupunând că regina `i` se plasează în poziţia `(i,j)` pe tabla de şah, **semnificaţia** acestei reprezentări apare în secvenţa [5.4.3.c].

```
-------------------------------------------------------------
/*Problema celor 8 regine - semnificaţia structurilor de
date*/                                         /*[5.4.3.c]*/
   x[i]=j     precizează locul j al reginei în coloana i
   a[j]=true  nici o regină nu ameninţă rândul j
   b[k]=true  nici o regină nu ameninţă diagonala /k
   c[k]=true  nici o regină nu ameninţă diagonala \k
-------------------------------------------------------------
```

- Se precizează că pe tabela de şah există 15 diagonale / (înclinate spre **dreapta**) şi 15 diagonale \ (înclinate spre **stânga**) (figura 5.4.3.a).
    - Caracteristica unei diagonale / este aceea că **suma** coordonatelor `i` şi `j` pentru oricare câmp care îi aparţine este o constantă.
    - Pentru diagonalele \ este caracteristic faptul că **diferenţa** coordonatelor `i` şi `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 / **sumele** `i+j` sunt cuprinse în domeniul `[2,16]`.
    - Iar pentru diagonalele \ **diferenţele** aparţin domeniului `[-7,7]`.

- Aceste considerente fac posibilă alegerea valorilor limitelor `b1,b2,c1,c2` pentru indicii tabelelor `b` şi `c` [5.4.3.b].
    - Una din posibilităţi este cea utilizată în continuare pentru care s-au ales `b1 = 2, b2 = 16, c1 = 0, c2 = 14`.
    - Pentru `b1` şi `b2` s-au ales chiar limitele intervalului în care iau valori **sumele indicilor**.

- Intervalul în care iau valori **diferenţele indicilor** a fost translatat cu valoarea 7 spre dreapta pentru a obţine valori **pozitive** pentru indicele de acces în tabela `c`.

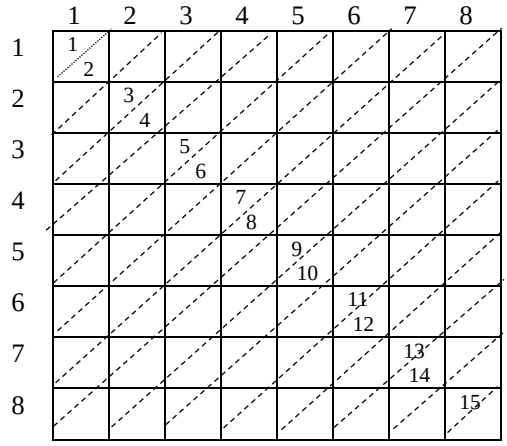



**Fig.5.4.3.a.** Tipuri de diagonale în problema celor 8 regine

- Cu alte cuvinte:
  - Accesul în tabloul `b` destinat evidenţei diagonalelor / se realizează prin `b[i+j]`.
  - Accesul în tabloul `c` destinat evidenţei diagonalelor \ prin `c[i-j+7]`.
  - Iniţial toate locaţiile tablourilor `a,b` şi `c` se poziţionează pe **`true`**.
- Cu ajutorul acestor reprezentări afirmaţia *****`plasează regina` pe poziţia `(i,j)`, `i` fiind coloana proprie, devine [5.4.3.d]:

```
-------------------------------------------------------------
/*plasează regina i pe poziţia (i,j)*/
 x[i]=j; a[j]=false; b[i+j]=false;          /*[5.4.3.d]*/
 c[i-j+7]=false
-------------------------------------------------------------
```

- În acelaşi context, afirmaţia *****`ia regina` apare rafinată în secvenţa [5.4.3.e]:

```
-------------------------------------------------------------
```

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

---

- Condiţia *`sigură` este îndeplinită dacă câmpul `(i,j)` destinaţie aparţine unui rând şi unor diagonale care sunt libere (**`true`**), situaţie ce poate fi exprimată de următoarea expresie logică [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 secvenţa [5.4.3.g].

---

```
#include <stdio.h>
/*găseşte o soluţie a problemei celor 8 regine*/

typedef unsigned boolean;                  /*[5.4.3.g]*/
#define true (1)
#define false (0)

int i; boolean q;
boolean a[8];         /*tabel disponibilitate rânduri*/
boolean b[15];        /*tabel disponibilitate diagonale /*/
boolean c[15];        /*tabel disponibilitate diagonale \*/
int x[8];             /*tabel plasare regine*/
void incearca(int i, boolean* q)
 {
  int j;
  j=0; /*iniţializează selecţie loc de plasare pentru
           regina i*/
  do {
    j=j+1;  /*selectează locul următor*/
    *q=false;
    if(a[j-1]&&b[i+j-2]&&c[i-j+7])  /*loc sigur*/
      {
        x[i-1]=j;       /*plasează regina i pe poziţia j*/
        a[j-1]=false;   /*indisponibilizează rândul j*/
        b[i+j-2]=false; /*indisponibilizează coloana /*/
        c[i-j+7]=false; /*indisponibilizează coloana \*/
        if(i<8) /*nu s-au plasat toate reginele*/
          {
           incearca(i+1,q); /*plasează regina următoare*/
           if(!*q) /*încercare nereuşită*/
            { /*şterge înregistrarea*/
             a[j-1]=true;   /*disponibilizează rândul j*/
             b[i+j-2]=true; /*disponibilizează coloana /*/
             c[i-j+7]=true; /*disponibilizează coloana \*/
            }
          }
         else  /*s-au plasat toate reginele*/
          *q=true; /*încercare reuşită*/
      }
  } while (!(*q||(j==8))); /*încercare reuşită sau nu mai
                             există locuri*/
```

```
  } /*incearca*/

int main(int argc, const char* argv[])
{ /*programul principal*/
  /*iniţializare tabel disponibilitate rânduri*/
  for(i=1;i<=8;i++)
    a[i-1]=true;
  /*iniţializare tabel disponibilitate diagonale /*/
  for(i=2;i<=16;i++)
    b[i-2]=true;
  /*iniţializare tabel disponibilitate diagonale \*/
  for(i=0;i<=14;i++)
    c[i]=true;
  /*plasează reginele*/
  incearca(1,&q);
  if(q) /*încercare reuşită - afişează plasarea reginelor*/
    for(i=1;i<=8;i++)
        printf("%i ", x[i-1]);
  printf("\n");
  return 0;
}
```

---

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

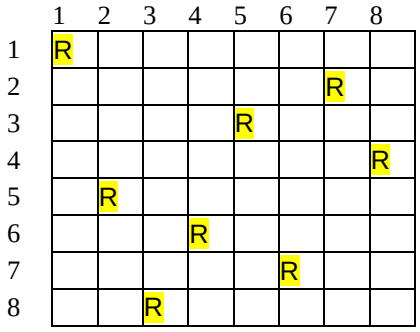



**Fig.5.4.3.b.** Soluţia problemei celor 8 regine

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

- Modelul de determinare a unei soluţii pentru o problemă de tip `(n,m)` poate fi uşor extins pentru a determina **toate soluţiile** unei astfel de probleme.
- Pentru aceasta este necesar ca **generarea paşilor** care construiesc soluţia 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 soluţie a fost găsită şi înregistrată, se trece la determinarea soluţiei următoare pe baza unui proces de selecţie sistematică până a la epuizarea **tuturor** posibilităţilor.
- În mod surprinzător, **găsirea tuturor soluţiilor unei probleme de tip (n,m)** presupune un algoritm mai **simplu** decât găsirea unei singure soluţii.

---

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

void incearcă_toate()                          /*[5.4.4.a]*/
    pentru(*toate posibilităţile de selecţie)
      daca(*selecţie acceptabilă)
        *înregistreaz-o ca şi curentă;
        daca(*soluţia incompletă)
            incearcă_toate *pasul următor;
          altfel
            *evidenţiază soluţia;
        *şterge înregistrarea curentă;
        □ /*daca*/
    □ /*pentru*/
/*incearcă*/
```

---

- Schema generală de principiu derivată din [5.4.2.a], care rezolvă această problemă apare în [5.4.4.a].
- Ca şi în cazul anterior, acest model principial va fi concretizat în **două variante**.
- (1) În prima **variantă**:
    - Procedura **incearca_toate_1** are drept parametru de apel **numărul pasului curent** şi realizează explorarea posibilităţilor în bucla interioară **FOR** [5.4.4.b].
    - Deoarece pentru evidenţierea tuturor posibilităţilor în fiecare pas trebuiesc parcurse toate valorile lui `posibilitate=[1,m]`, ciclul **repeat** a fost înlocuit cu unul **for**.

---

```
/* Rezolvarea unei probleme de tip (n,m). Determinarea
tuturor soluţiilor - Varianta 1 – pseudocod C*/

void incearca_toate_1(tip_pas i)              /*[5.4.4.b]*/
/*parametrul de apel este numărul pasului curent*/
  {
   tip_posibilitate posibilitate;
    for(posibilitate=1 pana la m )
      if(*acceptabilă){
          *înregistreaz-o ca şi curentă;
          if(i<n)
              incearca_toate_1(i+1);
            else
              *afisează soluţia;
          *şterge înregistrarea;
        }
```

```
  } /*incearca_toate_1*/
```

---

- (2) În cea de-a doua **variantă**:
  - Procedura **incearca_toate_2** are drept parametru de apel **o posibilitate de selecţie**.
  - Construcţia soluţiei se realizează apelând recursiv procedura succesiv pentru fiecare posibilitate în parte. [5.4.4.c].

---

```
/*Rezolvarea unei probleme de tip (n,m). Determinarea
tuturor soluţiilor - Varianta 2 - pseudocode C*/

void incearca_toate_2(tip_posibilitate posibilitate)
/*parametrul de apel este o posibilitate de selecţie*/
                                          /*[5.4.4.c]*/
  {
    if(*acceptabilă)
      {
        *înregistreaz-o ca şi curentă;
        if(*soluţie incompletă)
            {
              incearca_toate_2(posibilitate_1);
              incearca_toate_2(posibilitate_2);
                     ...
              incearca_toate_2(posibilitate_m);
            }
          else
            *afişează soluţia
        *şterge înregistrarea curentă
      }
  } /*incearca_toate_2*/
```

---

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

---

```
/*Problema celor 8 regine-determinarea tuturor soluţiilor*/

#include <stdio.h>                          /*[5.4.4.d]*/

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]);
    printf("\n");
  } /*afisare*/

void incearca_toate(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_toate(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_toate(1);
  return 0;
}
```

---

- Algoritmul prezentat anterior determină cele **92 de soluţii** ale problemei celor 8 regine.
- De fapt, din cauza simetriei există doar **12 soluţii distincte** care apar evidenţiate 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.** Soluţiile problemei celor 8 regine

### 5.4.5. Problema relaţiilor stabile

- **Specificarea** problemei **relaţiilor stabile**:
  - Se dau **două mulţimi disjuncte** `A` şi `B` de cardinalitate egală `n`.
  - Se cere să se găsească o mulţime de `n` perechi `<a,b>` astfel încât `a ∈ A` şi `b ∈ 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 relaţiei stabile**" ("**stable marriage rule**").
  - Se presupune că ia fiinţă un **ansamblu de dansuri** pentru care s-au înscris băieţi şi fete în număr egal (`n`).
  - În general fiecare băiat şi fiecare fată are **preferinţe distincte** faţă de partenerul său.
  - În baza acestor **preferinţe** se constituie `n` **perechi** de dansatori.
  - Dacă în această repartizare există un băiat şi o fată care **nu formează o pereche**, **dar** care se **preferă reciproc** faţă de actualii parteneri repartizaţi lor, se spune că **repartizarea este instabilă**.
  - Dacă **nu** există astfel de perechi, **repartizarea** se numeşte **stabilă**.
- Această situaţie se referă la o serie de probleme asemănătoare din viaţa de toate zilele în care atribuirile trebuiesc executate în raport cu anumite **preferinţe**: căsătorie, alegerea unei facultăţi de către un candidat, alegerea unei meserii, alegerea unui loc de muncă, selecţia donatorilor de organe, selecţia rezidenţilor pentru spitale, etc.
- În cazul de faţă problema va fi oarecum **simplificată** întrucât se presupune că **lista de preferinţe** este un **invariant** şi 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 schiţa algoritmului de rezolvare întrucât aceasta este evident o **problemă** de tip `(n,m)` căreia îi trebuiesc determinate **toate soluţiile**.
- 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 **băieţii** şi de a le distribui partenere, căutarea desfăşurându-se conform listei de preferinţe a băieţilor [5.4.5.a].

```
-------------------------------------------------------------
/*Problema relaţiilor stabile - pas rafinare 1*/

void incearcă(tip_baiat b);                    /*[5.4.5.a]*/
  {
    tip_ordin o;
    for(o=1;o<=n;o++)
      {
        *alege cea de-a o-a preferinţă a băiatului b;
```

```
        if(*acceptabilă)
          {
            *înregistrează perechea;
            if(b nu este ultimul băiat)
                incearcă(urmator(b));
              else
                *înregistrează setul stabil;
            *şterge perechea;
          } /*if*/
      } /*for*/
  } /*incearcă*/
```

---

- În continuare se vor preciza **structurile de date**.
  - Din raţiuni de claritate a programului se introduc **trei tipuri scalare** identice ca definiţie, dar cu nume diferite [5.4.5.b].
- **Datele iniţiale** sunt reprezentate prin două matrici care indică **preferinţele băieţilor** respectiv **preferinţele fetelor**.
  - `pref_baieti[b]` reprezintă lista de preferinţe a băiatului `b`, respectiv `pref_baieti[b,o]` este fata care ocupă poziţia `o` în lista băiatului `b`.
  - `pref_fete[f]` este lista de preferinţe a fetei `f`, iar `pref_fete[f,o]` este băiatul care reprezintă cea de-a `o`-a alegere a sa (fig.5.4.5.a).

---

```
/*Problema relaţiilor stabile - definirea structurilor de date */

typedef int tip_baiat;                          /*[5.4.5.b]*/
typedef int tip_fata;
typedef int tip_ordin;
tip_fata pref_baieti[n][n];  /*matrice preferinţe băieţi*/
tip_baiat pref_fete[n][n];   /*matrice preferinţe fete*/
```

---

- Un exemplu de date de intrare pentru problema relaţiilor stabile apare în figura 5.4.5.a.

| 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 relaţiei stabile

- **Rezultatul** execuţiei algoritmului va fi reprezentat în forma unui tablou de fete `x`, astfel încât `x[b]` reprezintă **partenera băiatului** `b`.
- Pentru egalitatea dintre între băieţi şi fete se mai introduce un tablou de băieţi `y`, astfel încât `y[f]` precizează **partenerul fetei** `f` [5.4.5.c].

```
-------------------------------------------------------------
/* Problema relaţiilor stabile - definirea structurilor de
date de ieşire*/
                                                  /*[5.4.5.c]*/
tip_fata x[n];  /*tabel de perechi ale băieţilor*/
tip_baiat y[n]; /*tabel de perechi ale fetelor*/
-------------------------------------------------------------
```

- Este evident că **nu** sunt necesare ambele tablouri, unul putându-se determina din celălalt conform relaţiilor: `x[y[f]]=f` şi `y[x[b]]=b` care sunt valabile pentru toate perechile constituite.
    - Deşi valorile lui `y[f]` pot fi determinate printr-o simplă baleere a lui `x`, totuşi `y` va fi păstrat, pe de o parte din raţiuni de **echitate** iar pe de altă parte pentru **claritatea** şi **eficienţa** programului generat.
- Informaţiile cuprinse în `x` şi `y` servesc la determinarea **stabilităţii** setului de perechi de dansatori care se construieşte.
    - Deoarece acest set este construit pas cu pas, generând câte o pereche căreia i se verifică imediat stabilitatea, `x` şi `y` sunt necesari chiar înainte ca toate perechile să fie stabilite.
- Pentru a păstra **evidenţa candidaţilor** se utilizează două tablouri booleene auxiliare: `b_neales` respectiv `f_nealeasa` [5.4.5.d].

```
-------------------------------------------------------------
/* Problema relaţiilor stabile - definirea unor structuri de
date auxiliare */

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

- Semnificaţia câmpurilor celor două tablouri apare în aceeaşi secvenţă.
    - După cum se observă, `b_neales[b]=`**`false`** implică că `x[b]` este definit, deci băiatul `b` are pereche.
    - Iar `f_nealeasa=`**`false`** implică că `y[f]` este definit, deci fata `f` are pereche.
- Deoarece alcătuirea perechilor se face pornind de la băieţi care sunt parcurşi 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 relaţiei [5.4.5.e]:

```
-------------------------------------------------------------
```

**NOT** b_neales[k] ≡ k<b [5.4.5.e]

---

- Aceasta sugerează faptul că tabloul b_neales **nu** este necesar şi în consecinţă se va utiliza un singur tablou, adică f_nealeasa destinat fetelor.

- Condiţia *acceptabilă va fi rafinată prin intermediul condiţiilor f_nealeasa[f] şi ***stabilă**, unde condiţia ***stabilă** va fi rafinată ulterior.

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

---

```
/*Problema relaţiilor stabile – pas rafinare 2*/

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

---

- **Elementul esenţial** în acest context îl constituie **determinarea stabilităţii**.

  - O primă observaţie se referă la faptul că **stabilitatea** se stabileşte **prin definiţie** din **compararea ordinelor de preferinţe**.

- Ordinul unui băiat sau al unei fete **nu** stă direct la dispoziţie din datele utilizate până în acest moment. El poate fi determinat printr-o căutare relativ costisitoare în tablourile pref_baieti respectiv pref_fete.

- Deoarece în determinarea stabilităţii **stabilirea ordinului de preferinţă** este o operaţie 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 preferinţă 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** şi pot fi uşor determinate din matricele iniţiale pref_baieti şi pref_fete.

---

```
/* Problema relaţiilor stabile - definirea unor structuri de
date auxiliare */
```

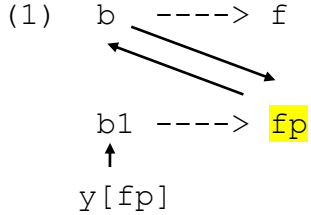
```
                                                    /*[5.4.5.g]*/
tip_ordin ofb[n][n]; /*ordinul de preferinţă pe care îl au
                       fetele în lista băieţilor*/
tip_ordin obf[n][n]; /*ordinul de preferinţă pe care îl au
                       băieţii în lista fetelor*/
```

---

- În continuare **stabilitatea** se determină în raport strict cu **definiţia sa iniţială**.
    - Se reaminteşte faptul că în cadrul algoritmului se încearcă alcătuirea perechilor, selectând la un moment dat băiatul `b∈[1,n]` şi fata `f`, unde `f=pref_baieti[b,o]` adică `f` este cea de-a `o`-a clasată în lista de preferinţe a lui `b`.
- În general, se presupune că stabilitatea este **predominantă** şi în continuare se încearcă găsirea **surselor de perturbaţie** posibile.
- În acest sens există **două posibilităţi**:
    - **Sursa** (1) Presupunând că băiatului `b` i-a fost repartizată fata `f`, ar putea exista o fată `fp`, preferată lui `f` de către `b` (`f` fiind perechea repartizată lui `b`), care la rândul ei îl preferă pe `b` perechii sale actuale `b1` [5.4.5.h (1)].
    - **Sursa** (2) Presupunând ca fata `f` a fost repartizată băiatului `b`, ar putea exista un băiat `bp` preferat lui `b` de către `f` (`f` fiind perechea repartizată lui `b`), băiat care la rândul său o preferă pe `f` perechii `f1` repartizate lui [5.4.5.h (2)].

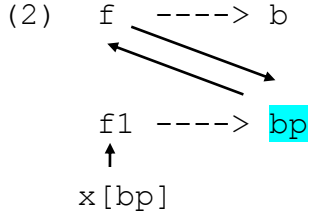
---

(1)




[5.4.5.h]

(2)




---

- Iniţial se urmăreşte **sursa** (1) de perturbaţie în vederea determinării **instabilităţii**:
    - Pentru toate fetele `fp` care au ordinul de preferinţă 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 `obf[fp,b]` şi `obf[fp,y[fp]]` pentru toate fetele `fp` preferate lui `f` de către `b`, adică pentru toate `fp=pref_baieti[b,i]` astfel încât `1≤i<o`, unde `o` este ordinul de preferinţă 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 preferinţă** cu valoare mai **mare** înseamnă o preferinţă mai **slabă**.
- Rezultatul comparaţiei este variabila booleană `s` care materializează **stabilitatea**. Atâta vreme cât `s` rămâne **`true`** relaţia este **stabilă**.
- Procesul descris poate fi formulat printr-o simplă căutare liniară [5.4.5.i].

---

```
/*Verificarea sursei 1 de perturbaţie*/

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

---

- În continuare se urmăreşte **sursa** (2) de perturbaţie:
  - Pentru toţi băieţii `bp` care au ordinul de preferinţă în lista fetei `f` mai mic ca şi 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 investigaţi toţi băieţii preferaţi `bp=pref_fete[f,i]` pentru `1≤i<obf[f,b]`.
    - În mod analog ca şi la cealaltă sursă de perturbaţie, este necesară compararea ordinele `ofb[bp,f]` şi `ofb[bp,x[bp]]`.
  - Vor trebui însă evitate comparaţiile care se referă la băieţii `bp` cărora încă **nu** le-au fost repartizate fete.
    - Testul care realizează acest obiectiv este `bp<b`, deoarece tuturor băieţilor care-l preced pe `b` li s-au repartizat deja perechi.
  - Rezultatul comparaţiei este tot variabila booleană `s` care atâta vreme cât rămâne **`true`** relaţia este **stabilă**.
  - Secvenţa de cod care implementează verificarea celei de-a doua surse de perturbaţie apare în [5.4.5.j].

---

```
/*Verificarea sursei 2 de perturbaţie*/

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]];
  }
```

---

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

---

```
/* Problema relaţiilor 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=of+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 obţin soluţiile stabile din figura 5.4.5.b.
- Programul are la bază un **algoritm cu revenire** din categoria celor trataţi în acest paragraf.
- **Eficienţa** sa depinde în primul rând de gradul de sofisticare al soluţiei de reducere a parcurgerii arborelui căutărilor.
    - Există şi alte metode mai eficiente decât cea propusă [MW71].
- Algoritmii de genul celui prezentat, care generează **toate** soluţiile posibile ale unei probleme sub incidenţa anumitor **constrângeri**, sunt adesea utilizaţi în selectarea uneia sau mai multor **soluţii optime** într-un sens precizat.

- În cazul de faţă poate interesa spre exemplu soluţia **cea mai favorabilă pentru băieţi**, pentru **fete**, sau pentru **toţi**.
- În tabela din fig.5.4.5.b apar în coloane specifice **suma ordinelor tuturor fetelor** alese din listele de preferinţe ale perechilor lor băieţi `(ofb)`şi **suma ordinelor tuturor băieţilor** aleşi din listele de preferinţe ale pe fetelor `(obf)` pentru fiecare soluţie în parte.
- Acestea sume se calculează conform relaţiilor [5.4.5.l].

---

$$ofb=\sum_{b=1}^{n} OFB[b,x[b]] \qquad obf=\sum_{b=1}^{n} OFB[x[b],b] \qquad [5.4.5.l]$$

---

| | x1 | x2 | x3 | x4 | x5 | x6 | x7 | x8 | ofb | obf |
|---|---|---|---|---|---|---|---|---|---|---|
| soluţia 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.** Soluţii ale programului `RelatieStabila`

- Soluţia cu cea mai **mică** valoare pentru `ofb` se numeşte **soluţia stabilă optimă pentru băieţi**
- Soluţia cu cea mai **mică** valoare pentru `obf` se numeşte **soluţia stabilă optimă pentru fete**.
    - Datorită strategiei de lucru adoptate în algoritm este evident că se obţine la început soluţia optimă pentru băieţi, iar cea pentru fete la sfârşit. În acest sens algoritmul favorizează băieţii.
    - Acest lucru poate fi însă uşor **evitat** dacă se schimbă rolul băieţilor şi al fetelor respectiv se interschimbă `PrefFete` cu `PrefBaieti` şi `OFB` cu `OBF`.
- Algoritmul „Stable marriage” a fost propus de David Gale şi de Loyd Shapley în anul 1962. După 50 de ani, în anul 2012, L.Shapley primeşte Premiul Nobel pentru acest algoritm împreună cu Alvin Roth, acesta fiind primul şi deocamdată sigurul premiu Nobel acordat pentru un algoritm.

### 5.4.6. Problema selecţiei 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 soluţie** 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 soluţiilor** unei probleme tip `(n,m)` (generalizarea plasării celor 8 regine).
    - În a treia etapă s-a abordat problema aflării **tuturor soluţiilor afectate de unele constrângeri** ale unei probleme tip `(n,m)` (relaţia stabilă).
- În continuare se propune aflarea **soluţiei optime**.
    - În acest scop se utilizează următoarea **strategie**: se generează **toate soluţiile posibile** iar pe parcursul procesului de generare este reţinută soluţia **optimă** într-un anumit sens.
- Presupunând că **optimul** este definit în termenii unei funcţii cu valori pozitive `f(s)`, algoritmul căutat derivă din modelul destinat aflării **tuturor soluţiilor** unei **probleme de tip** `(n,m)` [5.4.4.a] în care afirmaţia ***evidenţiază soluţia** se înlocuieşte cu secvenţa [5.4.6.a].

```
-------------------------------------------------------------
daca f(soluţie)>f(optim)  atunci optim= soluţie   [5.4.6.a]
-------------------------------------------------------------
```

    - Variabila `optim` înregistrează cea mai bună soluţie întâlnită până la momentul curent şi ea trebuie iniţializată 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 **soluţiei optime** este următorul.
    - Se consideră o mulţime de obiecte, numită **mulţime de bază**.
    - Se cere să se găsească **maniera optimă de selecţie** a unei **submulţimi de obiecte** care sunt supuse unor **constrângeri**.
    - Selecţionările care constituie **soluţii acceptabile**, se construiesc în mod gradat, examinând pe rând obiectele individuale ale **mulţimii de bază**.
- Prima rafinare a procedurii **`incearca`** [5.4.6.b], schiţează algoritmul de investigare al obiectelor mulţimii în vederea selecţiei, potrivit soluţiei.
    - 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 soluţia curentă.
        - (2) Fie cu **excluderea** sa din soluţia curentă.

- Din acest motiv utilizarea instrucţiunilor repetitive **do-while** sau **for** **nu**-şi are sensul, cele două situaţii specificându-se în mod explicit în schiţa de algoritm [5.4.6.b].

---

```
/*Determinarea selecţiei optime - schiţă de algoritm - pseudocod C*/

void incearca(tip_obiect i);                    /*[5.4.6.b]*/
  {
    if(*incluziunea este acceptabilă)
      {
        *include cel de-al i-lea obiect;
        if(i<n)
            incearca(i+1);
          else
            *verifică optimalitatea;
        *elimină al i-lea obiect;
      }
    if(*excluziunea este acceptabilă)
      if(i<n)
          incearca(i+1);
        else
          *verifică optimalitatea;
  } /*incearca*/
```

---

- Se presupune că obiectele mulţimii de bază sunt numerotate cu $1, 2, \ldots, n$.
    - Pornind de la acest model se obţin $2^n$ mulţimi selectate posibile.
- Este necesar însă să fie introduse **criterii** corespunzătoare care **să reducă** cât mai substanţial **numărul candidaţilor investigaţi** având în vedere faptul că se caută de fapt **selecţia optimă**.
    - Pentru lămurirea acestor aspecte, **modelul** anterior va fi **particularizat** după cum urmează.
- **Specificarea problemei** selecţiei optime:
    - Se consideră o **mulţime de bază** formată din $n$ obiecte $A_1, A_2, \ldots, A_n$.
    - Fiecare obiect $A_i$ este caracterizat prin **greutatea** sa $g_i$ şi prin **valoarea** sa $v_i$.
    - **Se cere** să se genereze **setul optim de obiecte** constând dintr-o selecţie a celor $n$ obiecte, selecţie care este afectată de una sau mai multe **constrângeri**.
        - Se defineşte drept **set optim** acel set care are **cea mai mare sumă a valorilor componentelor**.
        - Se precizează că **constrângerea** se referă la **limitarea sumei greutăţilor componentelor**.
- Aceasta este bine cunoscută **problemă a călătorilor** (**knapsack problem**), care îşi pregătesc bagajele selectând lucruri dintr-o mulţime $n$, astfel încât:
    - **Valoarea** lor (de întrebuinţare) să fie **maximă**.
    - **Greutatea** lor totală să **nu** depăşească o anumită **limită**.

- **Rezolvarea problemei** începe cu etapa stabilirii **structurilor de date** [5.4.6.c].

```
-------------------------------------------------------------
/* Determinarea selecţiei optime - definirea structurilor de
date */

enum {n=...};                                    /*[5.4.6.c]*/
typedef int *set;

typedef struct /*structura unui obiect*/
{
  int g;        /*greutate obiectului*/
  int v;        /*valoarea obiectului*/
} tip_obiect;

tip_obiect obiecte[n]; /*tabloul de obiecte*/

int limg;      /*limita de greutate*/
int vtot;      /*valoarea totală a tuturor celor n obiecte*/
int vmax;      /*valoarea selecţiei optime*/
set s;         /*selecţia curentă*/
set sopt;      /*selecţia optimă*/
-------------------------------------------------------------
```

- Variabila `limg` precizează **greutatea limită** impusă selecţiei.
- Variabila `vtot` precizează **valoarea totală** a celor `n` obiecte.
    - Ambele valori sunt **constante** pe parcursul întregului proces de selecţie.
- Variabila `s` reprezintă **selecţia curentă a obiectelor,** în care fiecare obiect este reprezentat prin numele său (indice).
- Variabila `sopt` memorează **selecţia optimă curentă.**
- Variabila `vmax` memorează **valoarea** selecţiei optime curente `sopt`.
- În continuare se vor preciza **criteriile de acceptare** ale unui obiect în selecţia curentă.
    - (1) În ceea ce priveşte **includerea:**
        - Un obiect este selectabil dacă odată adăugat selecţiei curente, aceasta se încadrează în **toleranţa de greutate.**
    - (2) În ceea ce priveşte **excluderea:**
        - Criteriul de acceptare al excluderii (cel care permite continuarea construirii selecţiei curente), este acela conform **căruia valoarea potenţială finală** `vp` la care s-ar putea ajunge **fără** a **include** candidatul curent în selecţie, **nu este mai mică** decât optimul întâlnit până în prezent.
        - **Neincluderea candidatului curent** reprezintă practic **excluderea** sa din selecţie.
        - În cazul în care valoarea `vp` curent determinată este mai **mică** decât optimul curent, procesul de căutare **poate** conduce şi la alte soluţii, **singur** însă **nu** la una **optimă**, deci căutarea în pasul curent **nu** este fructificabilă, ca atare este abandonată.

- În baza acestor **două** condiţii se precizează **elementele relevante** care urmează să fie determinate în fiecare pas al procesului de selecţie:
    - (1) **Greutatea totală** `gt` a selecţiilor executate până în prezent.
        - Variabila `gt` se iniţializează cu valoarea 0, căreia i se adaugă de fiecare dată greutatea obiectului inclus în selecţia curentă.
        - Condiţia `*incluziunea este acceptabilă` este ca noua greutate totală `gt` să **nu** depăşească limita de greutate `limg` [5.4.6.d].
    - (2) **Valoarea potenţială** `vp` care mai poate fi atinsă de către selecţia curentă `s`.
        - Iniţial `vp` ia valoarea maximă posibilă `vtot` obţinută prin însumarea valorilor tuturor celor `n` obiecte ale mulţimii.
        - La fiecare excludere din `vp` se scade valoarea obiectului exclus.
        - Condiţia `*excluziunea este acceptabilă` este adevărată dacă noua valoare a lui `vp` rămâne mai mare ca `vmax` unde `vmax` este valoarea corespunzătoare **soluţiei optime** determinate până în prezent [5.4.6.e].
            - Explicaţia este următoarea: chiar eliminând din selecţie obiectul curent, valoarea potenţială la care s-ar mai putea ajunge este mai mare decât cea a soluţiei optime curente, adică selecţia curentă poate încă conduce la o soluţie optimă.
- Entităţile precizate `gt` şi `vp` sunt declarate ca parametri ai procedurii **`Incearca`**.

```
-------------------------------------------------------------
/*rafinarea afirmaţiei *incluziunea este acceptabilă*/
  gt+obiecte[i].g <= limg                    /*[5.4.6.d]*/
-------------------------------------------------------------
/*rafinarea afirmaţiei *excluziunea este acceptabilă*/
 vp-obiecte[i].v > vmax                      /*[5.4.6.e]*/
-------------------------------------------------------------
```

- Spre a evita reevaluarea periodică a diferenţei `vp-obiecte[i].v`, aceasta se notează cu `vp1`.
- Verificarea **optimalităţii** şi **înregistrarea soluţiei optime** apare în secvenţa [5.4.6.f].

```
-------------------------------------------------------------
/*verifică optimalitatea*/
if(vp>vmax)                                  /*[5.4.6.f]*/
  {
    /*avem un nou optim care se înregistrează*/
    sopt=s;
    vmax=vp;
  }
-------------------------------------------------------------
```

- Ultima secvenţă este bazată pe raţionamentul că valoarea potenţială devine valoare efectivă, de îndată ce au fost tratate toate cele `n` obiecte.
- **Programul integral** în varianta `Pascal` apare în [5.4.6.g] şi în varianta `C` în [5.4.6.h].

- În varianta `Pascal` implementarea simplă şi elegantă a incluziunii şi excluziunii se datorează utilizării tipului mulţime **`SET`** şi a operatorilor specifici definiţi în limbaj.
- În varianta C operaţiile cu mulţimi sunt implementate într-o manieră explicită în forma tipului **`set`** respectiv a a operatorilor **`inset`**`(int, set)`, **`setof`**`(int,...)`, **`join`**`(set,set)` şi **`difference`**`(set,set)`.

---

```
PROGRAM SelectieOptima;

  CONST n=10;                                          [5.4.6.g]
  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;
      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;{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.
```

---

```
/* Selecţia Optimă */
                                                    /*[5.4.6.h]*/
#include <limits.h>
#include <stdarg.h>
#include <stdlib.h>

enum { n =10};

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

typedef int* set;
#define eos INT_MIN

typedef struct{
  int g,v;
} tip_obiect;

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

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);
    incearca(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 operaţiilor cu mulţimi de întregi----*/

boolean inset(int x, set s)
 /*verifică apartenenţa lui x la mulţimea s*/
 {      int i=0;
    for(i=1;i<=s[0];i++)
       if(s[i]==x)
          return true;
    return false;
 } /*inset*/

set setof(int x,...)
 /*generarea unei mulţimi de elemente*/
 {
   va_list ap;
   set s;
   int i=0,j=0,t,f;
   if((s=malloc((x+1) * sizeof(int)))==NULL)
     {
       printf("eroare!");
       exit(1);
     } /*if*/
   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]++;
          } /*if*/
        } /*for*/
         va_end(ap);
      } /*if*/
   return s;
 } /*setof*/

set join(set s1,set s2)
 {  /*reuniunea a două mulţimi s1 şi 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*/
   for(j=1;j<=s2[0];j++)
     if(inset(s2[j],s) ==false )
      {
        s[i++]=s2[j];
        s[0]++;
       } /*if*/
   return s;
 } /*join*/

 set difference(set s1, set s2)
 {  /*diferenţa a două mulţimi s1 şi 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];
       } /*if*/
   return s;
 } /*difference*/
```

---

- În fig.5.4.6.c sunt listate rezultatele **execuţiei** programului cu **limita de greutate** cuprinsă în intervalul `10÷120` cu ratia 10.

- **Algoritmii** care se bazează pe **tehnica revenirii** şi care utilizează un **factor de limitare** care restrânge parcurgerea arborelui potenţial de căutare, se mai numesc şi algoritmi de tip "**înlănţuie şi 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 execuţiei programului `SelectieOptima`

## 5.5. Structuri de date recursive

### 5.5.1. Structuri de date statice şi dinamice

- În cadrul capitolului 1 au fost prezentate **structurile de date fundamentale** respectiv tipurile nestructurate **enumerare**, **standard predefinite** şi **subdomeniu** precum şi tipurile structurate **tablou**, **articol** şi **secvenţă**.
- 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** şi **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** şi **structurile de date aferente lor** se numesc **statice**.
- În practica programării însă există multe probleme care presupun **structuri ale informaţiei mai complicate**, a căror caracteristică principală este aceea că **se modifică structural** în timpul execuţiei 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 execuţiei programului** căruia îi aparţine, 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 **margine superioară** a volumului şi 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 funcţie de maniera de **execuţie** a **algoritmului**.
    - Cu alte cuvinte acest volum poate să **crească** sau să **descrească** în dependenţă de anumiţi **factori** cunoscuţi **numai** în **timpul rulării**.
    - În consecinţă **alocarea memoriei** la o **structură dinamică** are loc în timpul **execuţiei** programului.
- În general, orice **variabilă** declarată în partea de declarare a variabilelor, cu excepţia celor de tip secvenţă, reprezintă o **structură statică**, pentru care se rezervă o **zonă de memorie constantă**.
    - Prin declararea unei **variabile statice** se precizează **numele** şi **tipul** ei.
        - **Numele** variabilei este un identificator prin intermediul căruia programul, respectiv programatorul, poate face **referire** la această variabilă.
- Până în momentul de faţă, singurele structuri dinamice abordate au fost cele de tip **secvenţă**.
    - Datorită faptului că **secvenţele** se presupun înregistrate în memorii externe, ele se consideră structuri dinamice **speciale** şi **nu** fac obiectul prezentului paragraf.
- De fapt atât structurile **dinamice** cât şi cele **statice** sunt formate în ultimă instanţă din componente de **volum constant** care se încadrează într-unul din **tipurile cunoscute** şi 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 şi **nodurile** lor individuale se deosebesc de structurile statice prin faptul că ele **nu se declară** şi în consecinţă **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 şi 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 aplicaţii speciale impun definirea unui **tip special** de variabile numite **variabile indicator**.
    - Valorile acestor variabile **nu** sunt **date efective** ci ele **precizează** (indică) **locaţii de memorie** care memorează **date efective**.
- Cu alte cuvinte, **valoarea** unei astfel de **variabile indicator** reprezintă o **referinţă** 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** şi anume **tipul de date abstract indicator**.
- Descrierea de principiu a unui **tip de date abstract indicator** apare în [5.5.2.a].

---

**TDA Indicator**

[5.5.2.a]

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

**Notaţii:** *p,q* - variabile de TipIndicator;
*e* - variabila de TipIndicat;
*b* - valoare booleana.

**Operatori:**

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

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

***Memorează Indicator****(p,q)* - copiază indicatorul *p* în *q*;

***Memorează ValoareIndicată****(p,e)* - copiază valoarea lui *e* în zona (locaţia) indicată de *p*;

*e=****Furnizează ValoareIndicată****(p)* - funcţie care returnează valoarea memorată în zona (locaţia) indicată de *p*;

*b=****IndicatorIdentic****(p,q)* - funcţie booleană ce returnează valoarea true dacă *p≡q*, adică cele

```
        două variabile indicator indică aceeaşi locaţie
        de memorie.
-------------------------------------------------------------
```

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

### 5.5.2.2. Implementarea TDA Indicator cu ajutorul pointerilor

- După cum se cunoaşte, **variabilele pointer**, sunt variabile statice obişnuite 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 **semnificaţia** 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 consecinţă, 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 acelaşi lucru se realizează prin ataşarea unor caractere speciale numelui variabilei pointer în dependenţă de limbajul utilizat [5.5.2.2.a].

```
-------------------------------------------------------------
{Precizarea unei variabile indicate - varianta Pascal}
VariabilaIndicata ≡ VariabilaPointer^ (Pascal) {[5.5.2.2.a]}
-------------------------------------------------------------
//Precizarea unei variabile indicate - varianta C
variabila_indicata ≡ *variabila_pointer (în C) //[5.5.2.2.a]
-------------------------------------------------------------
```

- În plus, în limbajul C s-a definit operatorul **&** care se permite **determinarea adresei** unei variabile indicate şi 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 stabileşte o **legătură fixă** între orice **variabilă pointer** şi **tipul variabilei indicate**.
    - O **variabilă pointer** dată se referă în mod **obligatoriu** la o **variabilă indicată** de **un anumit tip**.
- Această restricţie măreşte **siguranţa** în programare şi constituie o deosebire netă între **variabilele pointer** definite în limbajele de nivel superior şi **adresele obişnuite** din limbajele de asamblare.
- Un **tip pointer** se defineşte 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 restricţie 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** şi se realizează cu ajutorul unor **operatori standard** specifici dependenţi de limbaj.
- Pentru exemplificare în secvenţele următoare se prezintă implementarea **TDA Indicator** în limbajele Pascal respectiv C.
    - Implementările sunt realizate prin intermediul unor operatori definiţi la nivelul limbajului.

---

```
{TDA Indicator - implementare Pascal}

TYPE TipPointer = ^TipIndicat;                {[5.5.2.2.d]}
VAR p,q: TipPointer;
    e: TipIndicator;
    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:=IndicatorIdentic(p,q)}
```

---

```
/*TDA Indicator - implementare C*/

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

    p=malloc(sizeof(tip_indicat));  /*New(p)*/
    free(p);        /*Dispose(p)*/
    p= q;           /*MemoreazaIndicator(p,q)*/
    *p= e;          /*MemoreazaValoareIndicata(p,e)*/
    e= *p;          /*e:=FurnizeazaValoareIndicata(p)*/
    b=(p==q);       /*b:=IndicatorIdentic(p,q)*/
```

---

### 5.5.2.3. Implementarea TDA Indicator cu ajutorul cursorilor

- Dacă în limbajele care definesc **tipul pointer** (referinţă), **implementarea TDA Indicato**r este imediată şi realizată chiar la nivelul limbajului, în limbajele care **nu** dispun de această facilitate sau în situaţii speciale, implementarea acestui tip de date se poate realiza cu ajutorul **cursorilor**.

- Un **cursor** este o **variabilă întreagă** utilizată pentru a indica o **locaţie** într-un **tablou** de **variabile indicate**.
    - Ca şi metodă de conectare, un **cursor** este perfect **echivalent** cu un **pointer** cu deosebirea că el poate fi utilizat în plus şi în limbajele care **nu** definesc tipul referinţă.
    - Interpretând valoarea unei **variabile întregi** ca şi un **indice** într-un **tablou**, în mod efectiv acea variabilă va **indica** locaţia respectivă din tablou.
    - **Variabila indicată** este conţinută în locaţia respectivă din **tablou**.
- Cu ajutorul acestei tehnici, pot fi implementate practic **toate structurile de date** care presupun **înlănţuiri**, 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-şi dezvolte proprii operatori de tip **`New - Dispose`** respectiv **`alloc-free.`**
    - De asemenea, programatorul trebuie să rezerve **static** un **spaţiu 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 referinţelor

- Programarea orientată spre obiecte a impus noi dezvoltări ale posibilităţilor de acces la elementele cu care programatorul operează în cadrul unui program.
- Astfel limbajele orientate spre obiecte cum ar fi C++ sau JAVA definesc aşa numitele **referinţe** care de fapt sunt o formă de implementare a **tipului indicator**.
    - O referinţă apare ca fiind **similară** în multe privinţe cu un pointer, dar de fapt **nu** este un pointer.
- O **referinţă** este un **alias** (un alt nume) pentru o **variabilă precizată**.
    - O **referinţă** este de fapt un **nume** care poate fi folosit în **locul** variabilei originale.
    - Deoarece este un **alias** şi **nu** un pointer, **variabila** pentru care este declarată referinţa trebuie să fie **specificată** în momentul declarării referinţei.
    - În plus spre deosebire de un pointer o **referinţă 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 **referinţă** pentru ea utilizând următoarea declaraţie [5.5.2.4.b] :

---

```
    long& ref_numar = numar;  //declarare referinţă
                                          //[5.5.2.4.b]
-------------------------------------------------------------
```

- În acest caz se poate spune că de fapt `ref_numar` este de tip "**referinţă la `long`**".
- Referinţa 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 referinţa anterior precizată, un pointer la aceeaşi 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 distincţie evidentă între utilizarea unui **pointer** şi utilizarea unei **referinţe**.
  - **Pointerul** este evaluat la **fiecare utilizare** şi în conformitate cu adresa pe care o conţine la momentul respectiv este accesată **variabila indicată**.
  - **Referinţa nu** este evaluată la fiecare utilizare. Odată **definită** ea **nu mai poate fi modificată**, lucru care nu este valabil şi pentru pointer.
  - **Referinţa** este complet **echivalentă** cu variabila la care se referă.
- La o analiză sumară referinţa **nu** pare a fi altceva decât o **notaţie alternativă** pentru o anumită **variabilă** deoarece ea se comportă în acest sens.
- În realitate adevăratele valenţe ale acestui concept sunt puse în evidenţă 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 **referinţele** acelor obiecte şi nu obiectele ca atare sau copii ale lor.
  - Acest lucru are o importanţă cu totul particulară în dezvoltarea de aplicaţii orientate spre obiecte.
  - Spre exemplu în limbajul JAVA **nu** există **pointeri** expliciţi nici aşa numita "aritmetică a pointerilor" ci doar **referinţe**, 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 funcţii care se apelează pe ele însele.
- În cadrul acestui paragraf se va prezenta **extinderea** acestei proprietăţi şi asupra **tipurilor de date** în forma aşa-numitelor "**structuri de date recursive**".
    - Prin analogie cu algoritmii, prin **structură de date recursivă** se înţelege o **structură** care are cel puţin o **componentă** de acelaşi tip ca şi **structura însăşi**.
    - Şi în acest caz, definiţiile 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 înlănţuită** a cărei definire formală apare în [5.5.3.a].

```
-------------------------------------------------------------
/*Exemplu 1 de structură de date recursivă - structura listă
înlănţuită*/

typedef struct                                  /*[5.5.3.a]*/
  {
    tip_info info;
    tip_lista urm;        /*incorect*/
  } tip_lista;
-------------------------------------------------------------
```

- 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 şi un articol cu trei componente, prima reprezentând numele persoanei, celelalte două fiind arborii genealogici ai părinţilor.
- Aceasta se exprimă formal astfel [5.5.3.b]:

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

typedef struct                                  /*[5.5.3.b]*/
  {
    char* nume;
    tip_genealogie tata, mama;       /*incorect*/
  } tip_genealogie;
-------------------------------------------------------------
```

- Datorită câmpurilor `urm` respectiv `tata` şi `mama`, care au acelaşi tip ca şi structura însăşi, tipurile definite sunt **recursive**.
- Se face **precizarea** că **definiţiile** de mai sus **nu sunt corecte**, deoarece se utilizează identificatorii `tip_lista` repectiv `tip_genealogie` înainte ca ce aceştia să fie **complet definiţi**.

- Cu alte cuvinte, încalcă **convenţia** 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 uşor că cele două tipuri definesc **structuri infinite**.
  - Aceasta este o consecinţă directă a **recursivităţii** 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 infiniţi**.
  - În cazul arborilor genealogici spre exemplu, urmărind suficient de departe ascendenţa oricărei persoane se ajunge la strămoşi despre care **lipseşte informaţia.**
- Ţinând cont de aceasta, se va modifica definiţia tipului `tip_genealogie` astfel:
  - Dacă se ajunge la o **persoană necunoscută** atunci structura va conţine **un singur câmp** notat cu `'XX'`.
  - În **orice alt caz** structura conţine **trei câmpuri** conform definiţiei anterioare.
- În figura 5.5.3.a se poate urmări o structură de `tip_genealogie` conform definiţiei modificate.

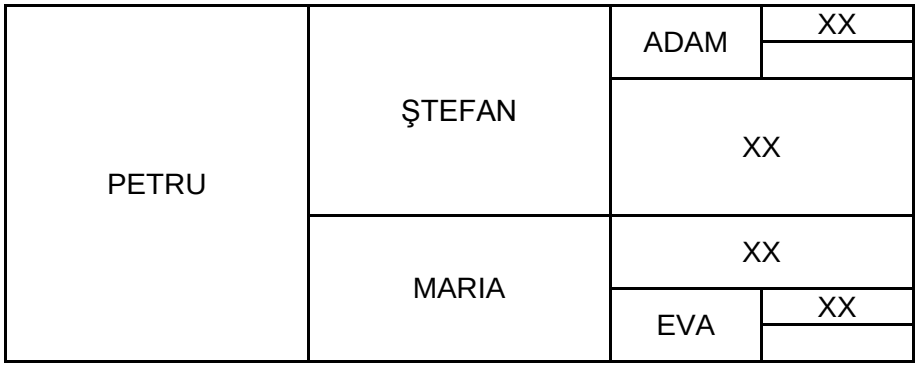



**Fig.5.5.3.a.** Exemplu de structură de date recursivă

- Cu privire la **tipurile de date recursive** se precizează în general că, pentru a evita structuri infinite, definiţia tipului trebuie să conţină o **condiţie** de care depinde prezenţa efectivă a componentei (sau a componentelor) recursive.
  - Ca şi în exemplul de mai sus, în anumite condiţii, componentele având acelaşi tip ca şi structura completă, pot să **lipsească**.
  - În accepţiunea acestor condiţionări pot exista **structuri recursive finite**.
- **Structurile recursive** pot fi implementate în limbajele de programare avansate **numai** în forma unor **structuri dinamice**.
- Prin **definiţie**, orice **componentă** care are **tipul identic cu structura completă**, se **înlocuieşte** cu un **pointer** care indică acea componentă.

- În secvenţele [5.5.3.c] apare pentru exemplificare definirea corectă a structurii recursive `tip_genealogie` în variantă C.

```
-------------------------------------------------------------
/*Structură de date recursivă - arborele genealogic al unei
persoane*/

struct genealogie                                /*[5.5.3.c]*/
  {
    char  *nume;
    struct genealogie  *tata, *mama;
  }
-------------------------------------------------------------
```

- După cum s-a precizat, în anumite condiţii, o componentă poate să lipsească dintr-o structură recursivă.
- În acest scop, mulţimea valorilor oricărui **tip referinţă** se extinde cu **referinţa vidă** (pointer nul) care nu se referă la nici o variabilă indicată.
    - Acest pointer se notează cu **`null`**, el aparţinând prin definiţie oricărui tip referinţă.
    - **Absenţa** unei **componente recursive** se va indica asignând pointerul referitor la această componentă cu **referinţa vidă**.
- Cu aceste precizări, **structura recursivă** din figura 5.5.3.a poate fi reprezentată ca în şi în figura 5.5.3.b.

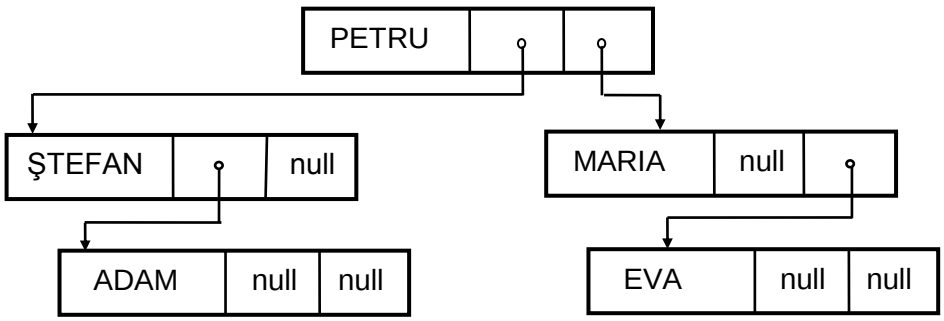



**Fig.5.5.3.b.** Implementarea unei structuri de date recursive

- Un alt exemplu de **structură recursivă** îl reprezintă **expresiile aritmetice**.
    - În acest caz **recursivitatea** rezultă din faptul că orice **expresie aritmetică** poate conţine 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 înţelege 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 secvenţa [5.5.3.d].

```
-------------------------------------------------------------
/*Structură de date recursivă - expresie aritmetică*/
                                                /*[5.5.3.d]*/
typedef struct tip_expresie* pointer_expresie;

typedef struct
  {
    char operator_;
    pointer_expresie operand1,operand2;
  } tip_expresie;
-------------------------------------------------------------
```

- În cadrul acestei definiţii se consideră că:

  - Dacă în câmpul **operator** se găseşte unul din caracterele `'+','-'`, `'*'` sau `'/'`, semnificând o **operaţie aritmetică**, atunci celelalte două câmpuri conţin câte un **pointer** la o altă **expresie**.

  - Dacă în câmpul **operator** se găseşte o literă, semnificând un **operand**, atunci celelalte două câmpuri vor conţine `NULL`.

- S-a arătat în &5.2.4, că orice **algoritm recursiv** poate fi înlocuit cu un **algoritm iterativ** echivalent.

- Această proprietate se păstrează şi pentru **structurile de date recursive** şi anume orice **structură recursivă** poate fi înlocuită cu o **structură secvenţă**.

- Astfel, o definiţie de tip recursiv de forma [5.5.3.e]:

```
-------------------------------------------------------------
/*Structură de date recursivă*/

typedef  struct   structura_recursiva      /*[5.5.3.e]*/
  {
    tip_continut continut;
    struct structura_recursiva  *urm;
  } tip_structura
-------------------------------------------------------------
```

- Este echivalentă şi înlocuibilă imediat cu tipul secvenţial [5.5.3.f]:

```
-------------------------------------------------------------
/*Varianta iterativă a structurii de date recursive
[5.5.3.e]*/
tip_structura = FILE OF tip_continut        /*[5.5.3.f]*/
-------------------------------------------------------------
```

- Se observă că o **structură recursivă** se poate înlocui imediat cu una **secvenţială** dacă şi numai dacă numele tipului recursiv apare în **propria definiţie recursivă** o **singură dată** la **sfârşitul** (sau la **începutul**) acesteia.

  - Această observaţie este valabilă şi pentru **procedurile** sau **funcţiile 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 secvenţială** prin **traversarea** arborelui asociat structurii într-o manieră ordonată specifică (spre exemplu în **in**, **pre** sau **postordine** respectiv **în adâncime** sau **prin cuprindere**).