5. Recursivitate

5.1. Introducere

- O definiție **recursivă** este acea definiție care se referă la un obiect care se definește ca parte a propriei sale definiri.

- Desigur o definiție de genul *"o floare este o floare"* care poate reprezenta în poezie un univers întreg, în știință în general și în matematică în special **nu** furnizează prea multe informații despre floare.

- O caracteristică foarte importantă a recursivității este aceea de a preciza o definiție într-un **sens evolutiv**, care evită circularitatea.

    - Spre exemplu o definiție **recursivă** este următoarea: *"Un buchet de flori este fie (1) o floare, fie (2) o floare adăugată buchetului"*.

        - Afirmația (1) servește ca și **condiție inițială**, indicând maniera de amorsare a definiției

        - Afirmația (2) precizează **definirea recursivă** (evolutivă) propriu-zisă.

    - Varianta **iterativă** a aceleași definiții este *"Un buchet de flori constă fie dintr-o floare, fie din două, fie din 3 flori, fie ... etc*".

    - După cum se observă, definiția recursivă este **simplă** și **elegantă** dar oarecum indirectă, în schimb definiția iterativă este directă dar greoaie.

- Despre un obiect se spune că este **recursiv** dacă el constă sau este definit prin el însuși.

- Prin definiție orice **obiect recursiv** implică **recursivitatea** ca și proprietate **intrinsecă** a obiectului în cauză.

- Recursivitatea este utilizată cu multă eficiență în **matematică**, spre exemplu în definirea numerelor naturale, a structurilor de tip arbore sau a anumitor funcții [5.1.a].

- Numerele naturale:

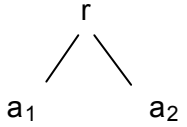
    (1) 1 este un număr natural;

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

- Structuri de tip arbore:

    (1) `r` este un arbore (numit arbore gol sau arbore cu un singur nod)

    (2) dacă a_1 și a_2 sunt arbori, atunci structura



este un arbore (reprezentat răsturnat). `[5.1.a]`

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

  (1) `0! = 1`

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

- Puterea recursivității rezidă în posibilitatea de a defini un set **infinit** de obiecte printr-o relație sau un set de relații **finit,** caracteristică evidențiată 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 apariția limbajelor de nivel superior care o implementează ca şi **caracteristică intrinsecă** (ALGOL, PASCAL, C, JAVA etc.).

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

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

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

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

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

    - În contrast cu iterația însă, în cadrul recursivității, condiția este verificată în **cursul** execuției programului (nu la sfârşitul ei ca la iterație)

    - În caz de rezultat nesatisfăcător, întreaga porțiune de program este apelată din nou ca **subprogram (procedură) a ei însăşi**, în particular ca procedură a porțiunii de program originale care încă **nu** şi-a terminat execuția.

    - În momentul satisfacerii condiției de revenire, se reia execuția programului apelant exact din punctul în care s-a apelat pe el însuşi.

    - Acest lucru este valabil pentru toate apelurile anterioare satisfacerii condiției.

- Utilizarea algoritmilor recursivi este potrivită în situațiile care presupun recursivitate (calcule, funcții sau structuri de date definite în termeni recursivi).

- În general, un program recursiv `P`, poate fi exprimat prin mulțimea $\wp$ care constă dintr-o serie de instrucțiuni fundamentale S_i (care nu-l conțin pe `P`) şi `P` însuşi.

$$P = \wp[S_i, P] \qquad [5.1.b]$$

  - Structurile de program necesare şi suficiente pentru exprimarea recursivității sunt **procedurile**, **subrutinele** sau **funcțiile** care pot fi apelate prin nume.

- Dacă o procedură `P` conține o referință directă la ea însăşi, se spune că este **direct recursivă**

- Dacă `P` conține o referință la o altă procedură `Q` care la rândul ei conține o referință (directă sau indirectă) la `P`, se spune că `P` este **recursivă indirect** (figura 5.1.a).

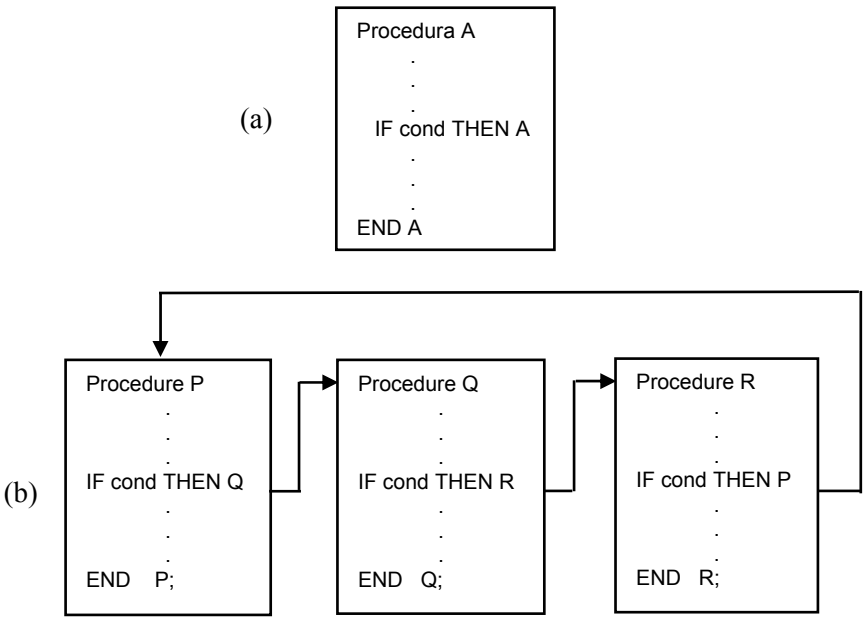


Fig.5.1.a. Recursivitate directă (a) şi recursivitate indirectă (b)

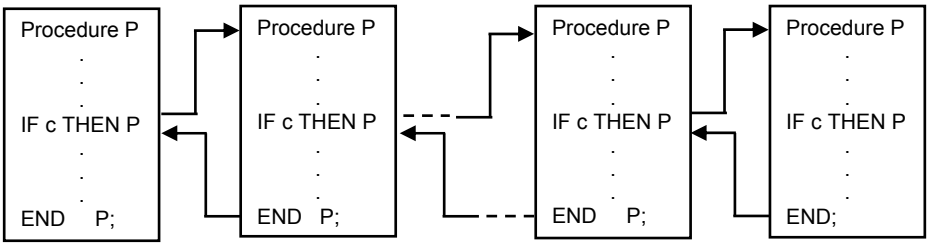


Fig.5.1.b. Model de execuție al unei proceduri recursive

- Intuitiv, execuția unei proceduri recursive este prezentată în figura 5.1.b.

    - Fiecare reprezentare asociată procedurii reprezintă o instanță de apel recursiv a acesteia.

    - Instanțele se apelează unele pe altele până în momentul în care condiția `c` devine falsă şi execuția respectivei instanțe se duce până la sfîrşit.

    - În acest moment se **revine** pe lanțul apelurilor în ordine inversă, continuând execuția fiecărei instanțe din punctul de întrerupere până la sfârşit, după cum indică săgețile.

    - Suprapunând imaginar toate aceste figuri peste una singură, se obține **modelul de execuție** al unei proceduri recursive.

- De regulă, unei proceduri i se asociază un **set de obiecte locale procedurii** (variabile, constante, tipuri şi proceduri) care sunt definite local în procedură şi care nu există sau nu au înțeles în afara ei.

    - Mulțimea 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 aceste variabile au acelaşi **nume** ca şi cele corespunzătoare lor din instanța 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 existență al identificatorilor.

    - **Regula** este următoarea:

        - **Identificatorii** se referă întotdeauna la setul cel mai **recent** creat de variabile, adică la contextul aflat în **vârful stivei**.

        - Aceaşi regulă este valabilă şi pentru **parametrii de apel** ai procedurii care sunt asociați prin definiție setului de variabile.

- Pe măsură ce se realizează apeluri recursive, **contextul** fiecărui apel este **salvat** în **stivă**, iar pe măsură ce execuția unei instanțe s-a încheiat, se **restaurează** contextul instanței apelante prin eliminarea contextului aflat în vârful stivei.

- Acest **mecanism** de implementare a recursivității este ilustrat în figura 5.1.c.

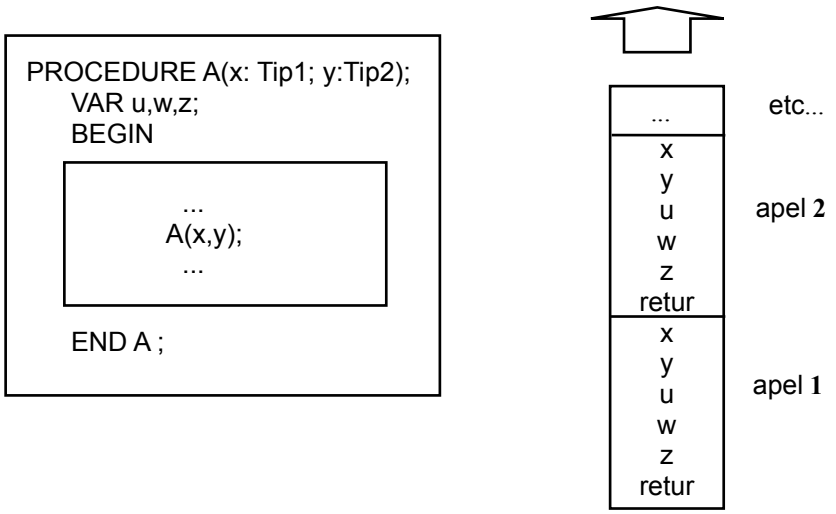


Fig.5.1.c. Mecanism pentru implementarea recursivității

- Ca şi în cazul structurilor repetitive, procedurile recursive necesită evaluarea unei **condiții de terminare**, fără de care un apel recursiv conduce la o buclă de program **infinită**.

    - Astfel, orice apel recursiv al procedurii `P` trebuie supus controlului unei **condiții** `c` care la un moment dat devine neadevărată.

- În baza acestei observații, un algoritm recursiv poate fi exprimat cu mai mare acuratețe prin una din formulele [5.1.e] sau [5.1.f]:

`P ≡ IF c THEN` p`[S_i,P]` `[5.1.e]`

`P ≡` p`[S_i,IF c THEN P]` `[5.1.f]`

- **Tehnica de bază** în demonstrarea **terminării** unei iterații (repetiții) **constă** în:
  - (1) A **defini** o funcție `f(x)` (unde `x` este o variabilă din program), astfel încât **condiția** `f(x)< 0` să implice satisfacerea condiției de terminare a iterației
  - (2) A **demonstra** că `f(x)` descreşte pe parcursul execuției iterației 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 condiția `c` cu `n > 0` , se garantează **terminarea**.
  - Acest lucru se poate exprima **formal** cu ajutorul următoarelor scheme de program [5.1.g,h]:

`P(n) ≡ IF n > 0 THEN` p`[S_i,P(n-1)]` `[5.1.g]`

`P(n) ≡` p`[S_i,IF n > 0 THEN P(n-1)]` `[5.1.h]`

- De regulă în aplicațiile practice trebuie demonstrat **nu** numai că **adâncimea recursivității 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 instanța apelantă.
- Neglijarea acestor aspecte poate avea drept consecință **depăşirea** spațiului de memorie alocat stivei sistem, greşeală frecventă în contextul utilizării procedurilor recursive.

5.1.1. Exemple de programe recursive simple

- Pentru exemplificare se furnizează în continuare trei exemple de proceduri recursive simple.
- **Exemplul 5.1.1.a.** Scopul programului furnizat în continuare ca exemplu, este acela de a ilustra **principiul de funcționare** al unei proceduri recursive.

- În cadrul programului se defineşte functia recursivă `revers:`
  - Funcția `revers` citeşte câte un caracter şi îl afişează.
    - În acest scop în cadrul procedurii se declară variabila locală `z:char.`
  - Funcția verifică dacă caracterul citit este `blanc:`
    - În caz **negativ** funcția se autoapelează recursiv
    - În caz **afirmativ** afişează caracterul `z` [5.1.1.a].

```
-----------------------------------------------------------------
/* Exemplu de program recursiv simplu  */
void revers(){
      char z;
      scanf("%c", &z);                        /*[5.1.1.a]*/
      if (z !=*" ")
             revers();
      printf("%c", z);
  }   /*Revers*/

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

- Execuția funcției `revers` conduce la afişarea caracterelor în ordinea în care sunt citite până la apriția primului blanc.
  - Fiecare autoapel presupune salvarea în stiva sistemului a contextului apelului care în situația de față constă doar din variabila locală `z`.
- Apariția primului caracter blanc presupune suprimarea apelurilor recursive ale funcției declanşând continuarea execuției ei, până la terminare, pentru fiecare din apelurile anterioare.
  - În cazul de față, continuarea execuției 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** a citirii lor.
  - Acest lucru se întâmplă deoarece fiecare **terminare** a execuției funcției 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 funcția `revers` furnizează cuvântul în cauză urmat de acelaşi cuvânt **scris invers**.
- **Exemplul 5.1.1.b.** Se referă la **traversarea** unei liste simplu înlănțuite [5.1.1.b].

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

typedef struct tipnod* tiplista;
typedef struct {
  int data;
```

```
  tiplista urm;
} tipnod;

void traversare(tiplista p)                          /*[5.1.1.b]*/
  {
    if (p!=0){
/*[1]*/     prelucrare(p->data);
/*[2]*/     traversare(p->urm);
      }
  }
-------------------------------------------------------------
```

- Algoritmul de traversare **nu** numai că este foarte **simplu** şi foarte **elegant**, dar prin simpla **inversare** a instrucțiunilor `prelucrare` şi `traversare` ([1] respectiv [2]) se obține parcurgerea în sens **invers** a listei în cauză.

- În acest ultim caz algoritmul se poate descrie astfel.

  - Se traversează mai întâi lista începând cu poziția curentă până la sfârşitul listei;

  - După ce s-a realizat această traversare se prelucrează informația din nodul curent;

- **Consecința** execuției algoritmului:

  - Informația conținută într-o anumită poziție `p` este prelucrată **numai** după ce au fost prelucrate **toate** informațiile corespunzătoare tuturor nodurilor care îi urmează lui `p`.

- **Exemplul 5.1.1.c.** 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;

  - Se cere să se **mute** discurile pe vergeaua C utilizând ca şi auxiliar vergeaua B;

  - Se va respecta următoarea **restricție**:

    - **Nu** se aşează niciodată un disc mai mare peste un disc mai mic (figura 5.1.1.a).

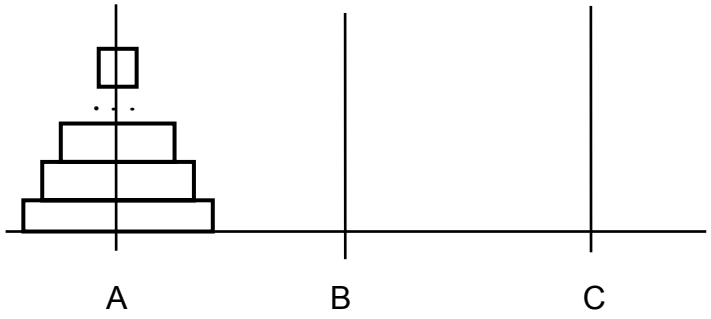


Fig.5.1.1.a. Turnurile din Hanoi

- 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 5.1.1.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).

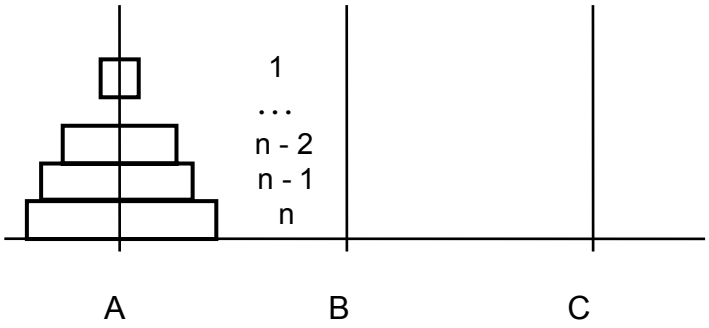


Fig.5.1.1.b. Generlizarea problemei Turnurilor din Hanoi

- Pornind de la acest **model**, o variantă imediată de implementare recursivă a rezolvării problemei este prezentată în secvenţa [5.1.1.a].

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


void mutadisc(tipvergele x/*de la*/,tipvergele y/*la*/)
  {
     ;/* muta discul  de la x la y*/
  }      /*MutaDisc*/

void turnurihanoi(int nrdiscuri, tipvergele a/*dela*/,
                                 tipvergele b/*la*/,
                                 tipvergele c/*prin*/);
  {
    if (nrdiscuri==1)
        mutadisc(a,c);                          /*[5.1.1.a]*/
```

```
      else
        {
          turnurihanoi(nrdiscuri-1, a, b, c);
          mutadisc(a,c);
          turnurihanoi(nrdiscuri-1, b, c, a);
        }   /*ELSE*/
  }      /*TurnuriHanoi*/
```

- Procedura `MutaDisc` realizează efectiv mutarea unui disc de pe o vergea sursă pe o vergea destinație.

- Procedura `TurnuriHanoi` realizează două auto-apeluri recursive şi un apel al procedurii `MutaDisc` conform modelului de generalizare prezentat mai sus.

5.2. Utilizarea recursivității

5.2.1. Cazul general de utilizare a recursivității

- **Algoritmii recursivi** sunt potriviți a fi utilizați atunci când **problema** care trebuie rezolvată sau datele care trebuiesc prelucrate sunt definite în **termeni recursivi**.

- Cu toate acestea, un astfel de mod de definire **nu** justifică întotdeauna faptul că utilizarea unui algoritm recursiv reprezintă cea mai bună alegere.

    - Mai mult, utilizarea recursivității în anumite situații nepotrivite, coroborată cu regia relativ ridicată a implementării şi execuției unor astfel de algoritmi, a generat în timp un curent de opinie **potrivnic** destul de vehement.

- Cu toate acestea recursivitatea rămâne o **tehnică de programare fundamentală** cu un domeniu de aplicabilitate foarte bine delimitat.

5.2.2. Algoritm recursiv pentru calculul factorialului

- Pentru a calcula valoarea factorialului se poate utiliza un subprogram funcție

    - Funcția poate fi utilizată direct ca şi constituent al unei expresii

    - Funcției i se poate asocia în mod explicit valoarea rezultată din calcule.

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

- Secvența în cauză care a fost redactată pentru calculul factorialului ca şi **întreg**, este **limitată** din punctul de vedere al dimensiunii maxime a lui `n` din rațiuni de reprezentare în calculator a numerelor întregi.

- Trecerea la varianta reală, eliberată de această constrângere este extrem de simplă.

- În figura 5.2.2.a apare reprezentarea intuitivă a **apelurilor** recursive ale funcției `Fact` pentru `n = 4`.

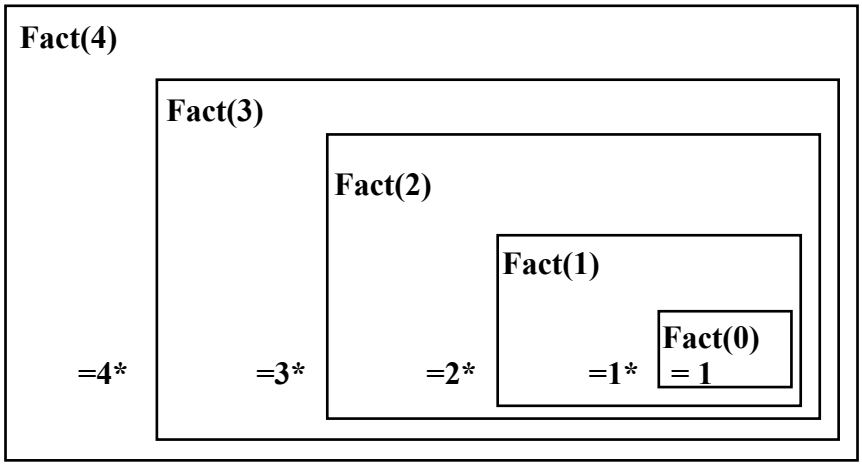


Fig.5.2.2.a. Apelurile ale funcției recursive `Fact` pentru `n = 4`

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

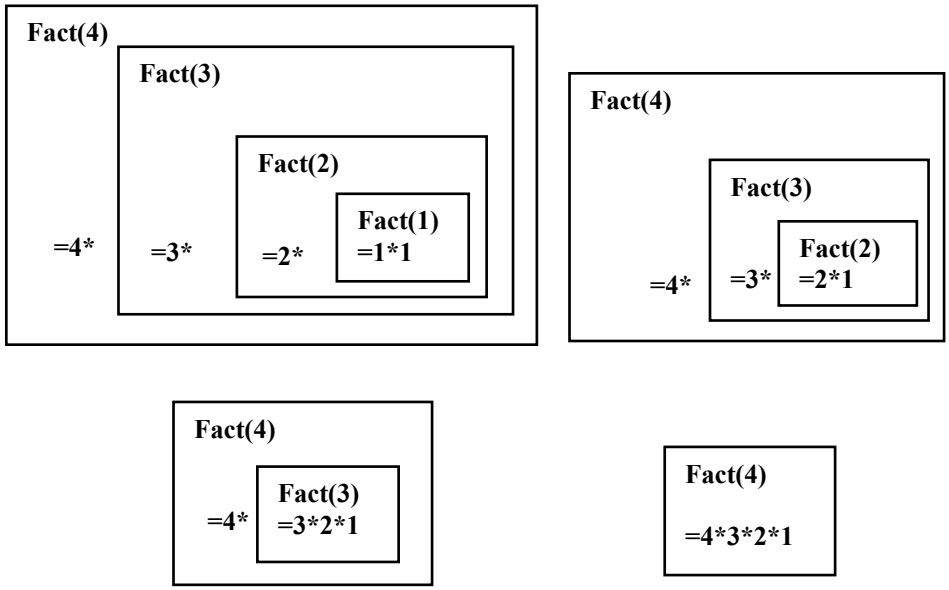


Fig.5.2.2.b. Rezolvarea apelurilor funcției recursive `Fact` pentru `n = 4`

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

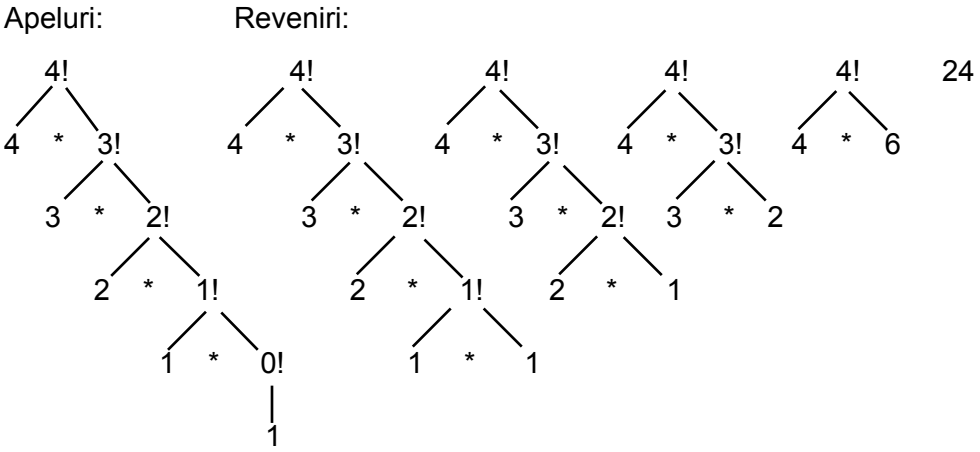


Fig.5.2.2.c. Arborele de apeluri al funcției `Fact(4)`

- În acest caz fiind vorba despre despre o funcție 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 consecință 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 funcției recursive `Fact`.

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

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

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

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

    - Sunt de fapt reprezentați în manieră comparativă, timpii de execuție pe un acelaşi sistem de calcul, ai celor doi algoritmi funcție de valoarea lui `n` .

    - După cum se observă, deşi ambii algoritmi sunt **liniari** în raport cu `n`, adică au performanța O(*n*) , algoritmul recursiv este de aproximativ 4 ori mai lent decât cel iterativ.

    - Expresiile analitice ale celor două reprezentări apar în [5.2.2.c][De84].

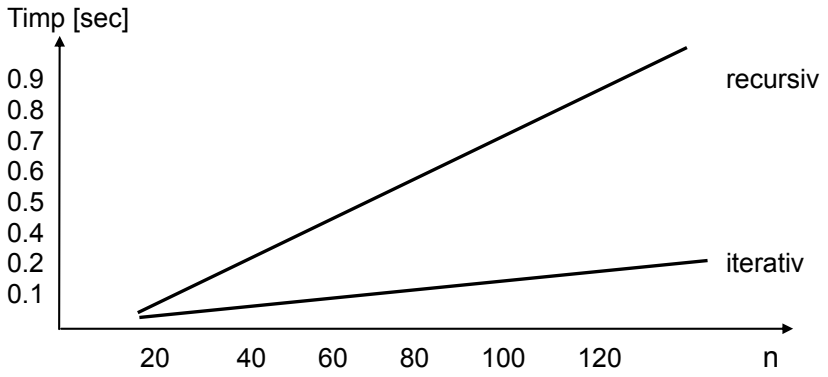


Fig.5.2.2.d. Profilul algoritmului factorial (variantele recursivă şi iterativă)

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

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

5.2.3. Numerele 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 $n > 0$

$Fib_1 = 1;\ Fib_0 = 0$ [5.2.3.a]

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

```
/* Calculul numerelor lui Fibonacci  */

int fib(int n)
  {
    int fib_result;
    if (n==0)  fib_result=0; else             /*[5.2.3.b]*/
    if (n==1)  fib_result=1; else
      fib_result=fib(n-1) + fib(n-2);
    return fib_result;
  }
/*-----------------------------------------------------------*/
```

- 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`.

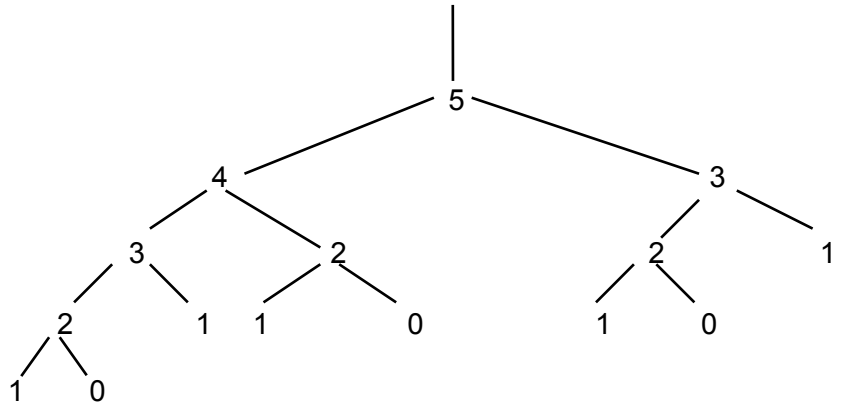


Fig.5.2.3.a. Arborele de apeluri al procedurii `Fib` pentru `n = 5`

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

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

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

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

5.2.4. Eliminarea recursivității

- 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 eficienței şi performanței se pun cu preponderență, se recomandă **evitarea** utilizării recursivității.
  - De asemenea se recomandă evitarea recursivității ori de câte ori stă la dispoziție o **rezolvare evidentă** bazată pe iterație.
- De fapt, implementarea recursivității pe echipamente nerecursive 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 instrucțiune a procedurii ("**tail recursion**").

    - În această situație, recursivitatea poate fi înlocuită cu o **buclă** simplă de iterație.

    - Acest lucru este posibil, deoarece în acest caz, **revenirea** dintr-un apel încuibat presupune şi **terminarea** instanței respective a procedurii, motiv pentru care contextul apelului **nu** mai trebuie restaurat.

- Astfel dacă o procedură `P(x)` conține ca şi **ultim pas** al său un apel la ea însăşi de forma `P(y)`

    - Acest apel poate fi înlocuit cu o instrucțiune de atribuire `x:=y` , urmată de reluarea (salt la începutul) codului lui `P` .

    - `y` poate fi chiar o expresie,

    - `x` însă în mod obligatoriu trebuie să fie o variabilă transmisibilă prin **valoare**, astfel încât valoarea ei să fie memorată într-o locație specifică apelului.

    - De asemenea `x` poate fi transmis prin **referință** dacă `y` este chiar `x` .

    - Dacă `P` are mai mulți parametri, ei pot fi tratați fiecare în parte ca `x` şi `y`.

- Această modificare este valabilă deoarece reluarea execuției 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 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].

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

    - Fiecare apel necesită **salvarea** în stiva gestionată de către utilizator a contextului instanței de apel,

    - Acest context trebuie restaurat de către utilizator din aceeaşi stivă la terminarea instanței.

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

- Activitatea de conversie în acest caz are un înalt grad de **specificitate** funcție de algoritmul recursiv care se dorește a fi convertit și este în general dificilă, laborioasă și îngreunează mult înțelegerea algoritmului.

- 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 formulați ca și proceduri recursive, lucru pus în evidență de algoritmii prezentați în cadrul acestui capitol și în capitolele următoare.

5.3. Exemple de algoritmi recursivi

5.3.1. Algoritmi care implementează definiții recursive

- **Exemplu. Algoritmul lui Euclid**
    - Permite determinarea **celui mai mare divizor comun** a două numere întregi date.
    - Se definește în manieră recursivă după cum urmează:
        - Dacă unul dintre numere este **nul**, c.m.m.d.c. al lor este celălalt număr;
        - Dacă nici unul din numere **nu** este nul, atunci c.m.m.d.c **nu** se modifică dacă se înlocuiește unul din numere cu restul împărțirii sale cu celălalt.
- Pornind de la această definiție recursivă se poate concepe un subprogram simplu de tip funcție pentru calculul c.m.m.d.c. a două numere întregi [5.3.1.a].

```
/* Implementarea algoritmului lui Euclid  */

int cmmdc(int m,int n)
  {
    int cmmdc_result;
    if (n==0)
        cmmdc_result=m;                          /*[5.3.1.a]*/
      else
        cmmdc_result=cmmdc(n, m%n);
    return cmmdc_result;
  }
```

- În figura 5.3.1.a apare urma execuției acestui algoritm pentru valorile 18 și 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.a. Urma execuției algoritmului lui Euclid

5.3.2. Algoritmi de divizare

- 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 și din soluțiile cărora se poate asambla simplu soluția problemei inițiale.

- (2) Se repetă **recursiv** pasul (1) până când subproblemele devin banale iar soluțiile lor evidente.

- O aplicație **tipică** a tehnicii divizării are structura recursivă prezentată în [5.3.2.a].

```
{Tehnica divizării - soluția recursivă}

PROCEDURE Rezolva(x);
  BEGIN
    IF  *x este divizibil în subprobleme THEN
         BEGIN                                        [5.3.2.a]
           *divide pe x în două sau mai multe părți:
              x1,x2,..,xk;
           Rezolva(x1); Rezolva(x2),...; Rezolva(xk);
           *combină cele k soluții parțiale într-o
              soluție pentru x
         END{IF}
       ELSE
         *rezolvă pe x direct
  END;{Rezolva}
```

```
/* Tehnica divizării - soluția recursivă */

void rezolva(x);
  {
    if  (*x este divizibil în subprobleme) THEN
         {                                          /*[5.3.2.a]*/
           /*divide pe x în doua sau mai multe parti:
               x1,x2,..,xk;*/
           rezolva(x1);
           rezolva(x2);
              ...
           rezolva(xk);
           /*combină cele k soluții parțiale într-o soluție
             pentru x*/
         }/*if*/
       else
         /*rezolva pe x direct*/;
  };
```

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

- **Analiza algoritmilor de divizare**.

  - Se presupune că timpul de execuție al rezolvării problemei de dimensiune `n` este `T(n)` .
  - În condițiile în care prin divizări succesive problema de rezolvat devine suficient de redusă ca dimensiune, se poate considera că pentru $n \le c$ (c constant), determinarea soluției necesită un timp de execuție constant, adică $O(1)$.
  - Se notează cu `D(n)` timpul necesar **divizării** problemei în subprobleme şi cu `C(n)` timpul necesar **combinării** soluțiilor parțiale.
  - Dacă problema inițială se divide în `k` **subprobleme**, fiecare dintre ele de dimensiune `1/b` din dimensiunea problemei originale, se obține următoarea formulă recurentă [5.3.2.b].

$$T(n) = \begin{cases} O(1) & \text{dacă } n \le c \\ k \cdot T(n/b) + D(n) + C(n) & \text{pentru } n > c \end{cases} \qquad [5.3.2.b]$$

- În continuare se prezintă câteva aplicații ale acestei tehnici.

- **Exemplul 5.3.2.a**.
    - Un prim exemplu de algoritm de divizare îl constituie metoda de sortare **Quicksort** (&.3.2.6).
    - În acest caz problema se divide de fiecare dată în **două subprobleme**, rezultând un arbore binar de apeluri.
    - Combinarea soluțiilor parțiale **nu** este necesară deoarece scopul este atins prin modificările care se realizează chiar de către rezolvările parțiale.

- **Exemplul 5.3.2.b.** Algoritm pentru determinarea extremelor valorilor componentelor unui vector.
    - În acest scop, poate fi proiectată o procedură `Domeniu(a,i,j,mic,mare)` care atribuie parametrilor `mic` şi `mare` elementul minim respectiv maxim al vectorului `a` din domeniul delimitat de indicii `i` şi `j` (`a[i]..a[j]`).
    - O implementare iterativă evidentă a procedurii apare în secvența [5.3.2.c] sub denumirea de `DomeniuIt`.

```
/* Algoritm pentru deteterminarea extremelor unui vector –
soluţia iterativă */
#define n 100    /*definim marimea tabloului*/
typedef int tiptablou[n-1];
void domeniuit(tiptablou const a, int i,int j, int* mic,int*
mare)
  {
     int k;
     *mic=a[i]; *mare=a[i];
     for( k=i+1; k <=j; k++)                /*[5.3.2.c]*/
      {
        if (a[k]>*mare) *mare=a[k];
        if (a[k]<*mic)  *mic=a[k];
      }
  }
```

- 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 `Domeniu` în termenii numărului de comparații 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** ține cont de faptul că orice element luat în considerare drept candidat pentru minim (care apare în succesiunea de valori a lui `mic`) nu poate fi niciodată candidat pentru maxim, exceptând condiția de inițializare şi reciproc.
- Astfel algoritmul risipeşte un efort considerabil examinând fiecare element de două ori. Această risipă se poate evita.

- Soluția **recursivă**:
    - Aplicând tehnica divizării, se împarte tabloul în două părți
    - Se compară valorile minime şi maxime ale subtablourilor rezultate, 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 soluția este evidentă.

- O ilustrare a acestei tehnici apare în procedura recursivă `Domeniu` [5.3.2.d].

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

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

void domeniu(tiptablou const a, int i,int j,int* mic,int* mare)
  {
    int mijloc,mic1,mare1,mic2,mare2;
    if (j<=i+1)      /*tablou de dimensiune 1 sau 2*/
        {
          if (a[i-1]<a[j-1])
              {
                *mic=a[i-1]; *mare=a[j-1];
              }
            else
              {                                    /*[5.3.2.d]*/
                *mic=a[j-1]; *mare=a[i-1];
              }
        }  /*IF*/
      else
        {
          mijloc=(i+j) / 2;
          domeniu(a,i,mijloc,&mic1,&mare1);
          domeniu(a,mijloc+1,j,&mic2,&mare2);
          if (mare1>mare2)
                *mare=mare1;
             else
                *mare=mare2;
          if (mic1<mic2)
                *mic=mic1;
             else
                *mic=mic2;
        }  /*ELSE*/
  }   /*Domeniu*/
-------------------------------------------------------------------
```

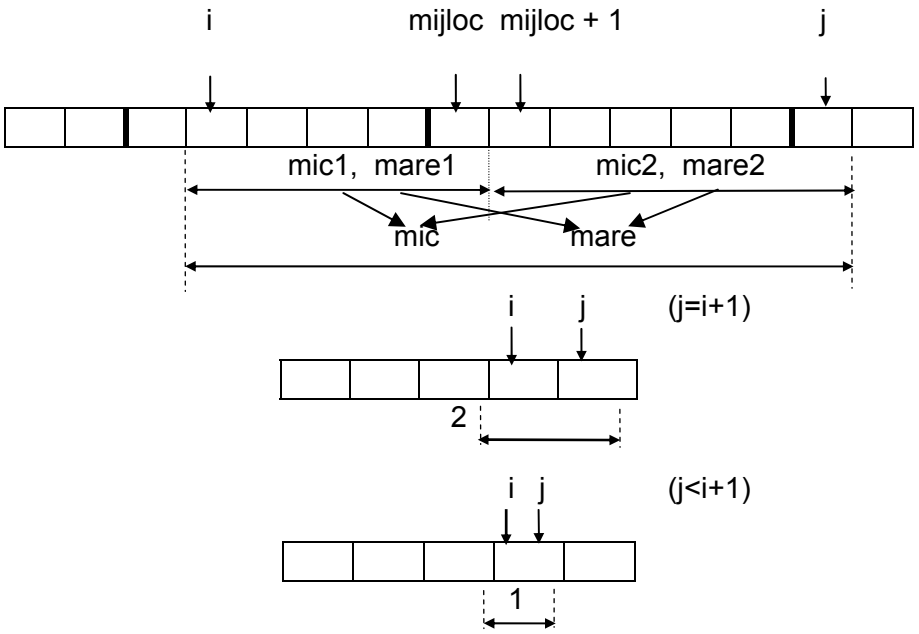


Fig.5.3.2.a. Funcționarea de principiu a procedurii `Domeniu`.

- Se constată **analogia** evidentă cu structura de principiu prezentată în secvența [5.3.2.a].

- În figura 5.3.2.a. apare **reprezentarea grafică** a modulului de lucru al procedurii `Domeniu`.
- **Urma execuției** procedurii pentru `n = 10` apare în figura 5.3.2.b,
    - În figură apar precizate limitele domeniilor pentru fiecare dintre apelurile procedurii.
    - În chenar sunt precizate domeniile de lungime 1 sau 2 care presupun comparații directe.
    - După cum se observă, arborele de apeluri este un **arbore binar** deoarece se execută două apeluri recursive din corpul procedurii.
- Dacă se realizează o analiză mai aprofundată a modului de implementare se constată că:
    - Contextele apelurilor se **salvează** în stiva sistem în ordinea rezultată de parcurgerea în preordine a arborelui de apeluri
    - Contextele se **extrag** din stivă conform parcurgerii în postordine a acestuia.
    - **Restaurarea** contextului fiecărui apel face posibilă parcurgerea ordonată a tuturor posibilităților evidențiate de arborele de apeluri.

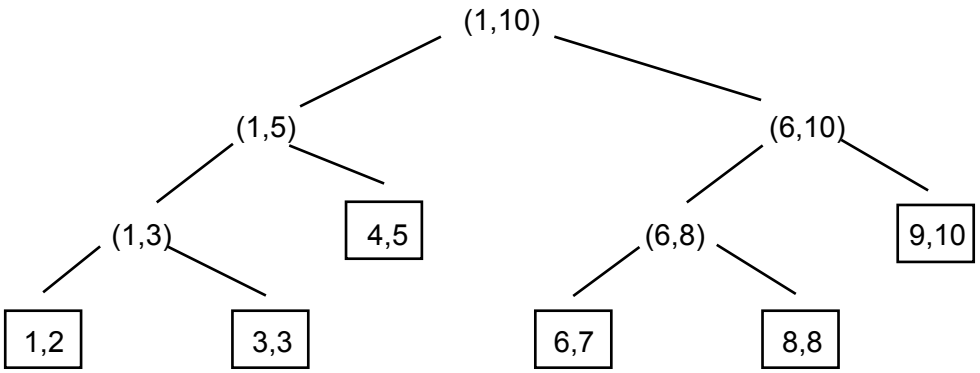


Fig.5.3.2.b. Arbore de apeluri al procedurii `Domeniu` pentru `n = 10`

- 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 **comparații** ale elementelor tabloului, el este mai eficient decât algoritmul iterativ.
    - Pornind de la observațiile:
        - (1) Procedura include comparații directe numai pentru subtablourile scurte (de lungime 1 sau 2)
        - (2) Pentru subtablourile mai lungi se procedează la divizarea intervalului comparând numai extremele acestora,
    - În [AH85] se indică o valoare aproximativă pentru cost (număr de comparații directe) egală cu `(3/2)n-2` unde `n` este dimensiunea tabloului analizat.

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 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** soluția integrală.

- În această categorie pot fi încadrați algoritmul pentru calculul factorialului şi algoritmul pentru rezolvarea problemei Turnurilor din Hanoi.
- Se atrage atenția că spre deosebire de algoritmii cu revenire (&5.4), în acest caz **nu** se pune problema revenirii în caz de nereuşită, element care încadrează acest tip de algoritmi într-o categorie separată.
- În continuare se prezintă un **exemplu** de algoritm de reducere.

- **Exemplul 5.3.3.a.** Algoritm pentru **determinarea permutărilor primelor `n` numere naturale.**
  - Se dă o secvență de numere naturale
  - Se cere să se determine toate permutările care se pot construi cu această secvență de numere
  - Pentru determinarea **permutărilor** se poate utiliza **tehnica reducerii**.
  - Principiul este următorul:
    - Pentru a obține permutările de `n` elemente este suficient să se fixeze pe rând câte un element 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 `Permut(k)` care realizează permutarea primelor `k` numere naturale.
- Schița de principiu a acestui algoritm apare în secvența [5.3.3.a.]

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

Procedura Permuta(k:integer);
    dacă k=1 atunci
       *afişează tabloul (s-a finalizat o permutare)
    altfel
      pentru i=1 la k execută                    [5.3.3.a]
        *interschimba pe a[i] cu a[k];
        Permuta(k-1); {apel recursiv}
        *interschimba pe a[i] cu a[k] {refacere situatie}
```

- Numerele se presupun memorate în tabloul `a[i](i=1,2,...,n)`.
- Dacă `k` ≠ `1`, atunci **fiecare** dintre elementele tabloului situate pe poziții **inferioare** lui `k` sunt aduse (fixate) pe poziția `k` prin **interschimbarea** pozițiilor `i` şi `k` ale tabloului `a` în cadrul unei bucle **`FOR`** pentru `i=1,2,...,k`.
- Inițial `k` ia valoarea `n` pentru permutarea a `n` numere.
- Pentru fiecare schimbare (fixare) se **apelează** recursiv rutina cu parametrul `k-1`, după care se **reface** starea inițială reluând în sens invers interschimbarea pozițiilor `i` şi `k`.
  - Acest lucru este necesar, deoarece fixarea elementului următor presupune **refacerea** stării inițiale în vederea parcurgerii în mod ordonat, pentru fiecare element în parte a **tuturor** posibilităților.
- În momentul în care `k` = 1, se consideră **terminat** un apel şi se afişează tabloul `a` care conține o permutare.
- Aceasta este **condiția** care limitează adâncimea apelurilor recursive determinând revenirea în apelurile anterioare.

- În secvența următoare apare o varinată de implementare a acestui algoritm

---

```
/* Exemplu de implementare a algoritmului pentru determinarea
permutărilor a  n numere naturale */
```

```
void permuta(int k)
  {
      int i,x;
      if (k==1)
      afiseaza;
      else
        {
         for( i=1; i <=k; i ++)
           {                                          /*[5.3.3.b]*/
            x=a[i]; a[i]=a[k]; a[k]=x;
            permuta(k-1);
            x=a[i]; a[i]=a[k]; a[k]=x;
           }
        }
  }
```

- În figura 5.3.3.a este reprezentat **arborele de apeluri** al procedurii `Permuta(4)` pentru permutările obținute prin fixarea primului element

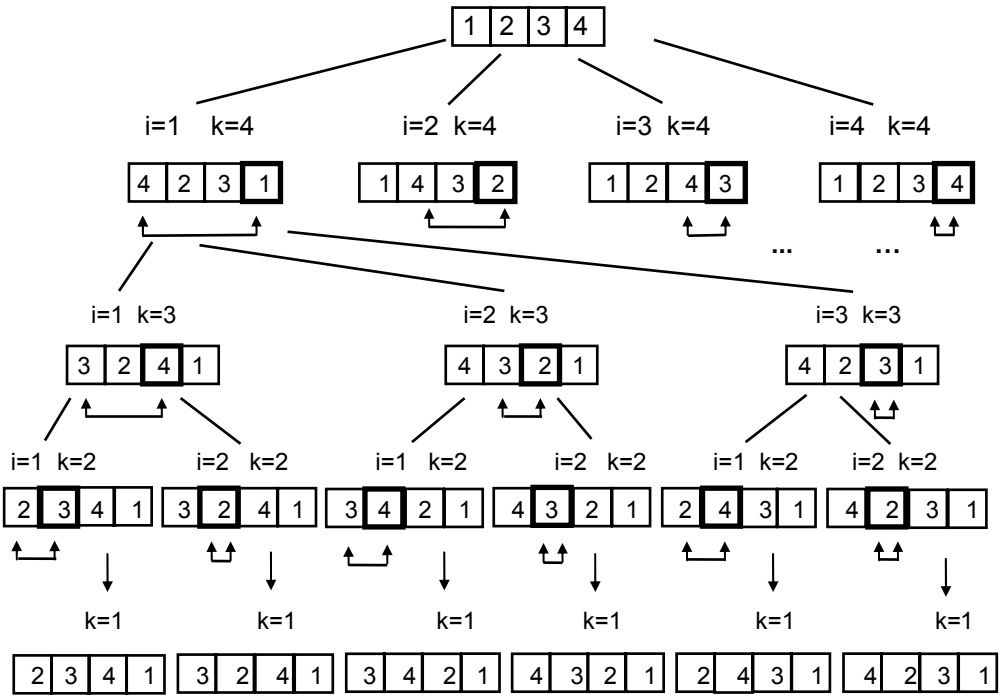


Fig.5.3.3.a. Arborele de apeluri pentru `Permuta(4)`

- 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.
- Înălțimea arborelui de apeluri este egală cu n.

5.3.4. Algoritmi recursivi pentru determinarea tuturor soluțiilor unor probleme

- Algoritmii recursivi au **proprietatea** de a putea evidenția în mod ordonat **toate posibilitățile** referitoare la o situație dată.
- Se prezintă în acest sens două exemple
  - Primul exemplu reliefează proprietatea de a evidenția în mod ordonat **toate** posibilitățile referitoare la o situație dată. (exemplul 5.3.4.a).
  - Cel de-al doilea exemplu **selectează** din mulțimea tuturor posibilităților evidențiate, pe cele care prezintă interes (exemplul 5.3.4.b).

- **Exemplul 5.3.4.a.** Algoritm pentru evidenţierea tuturor posibilităţilor 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 firului `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.a].

```
-------------------------------------------------------------
{Schita de principiu a algoritmului de tăiere a firului}

Procedura Taie(lungimeFir);
  dacă lungimeFir>1 atunci
     *se taie o bucată de lungime 1;
     Taie(lungimeFir-1);
     *se taie o bucată de lungime 2;
     Taie(lungimeFir-2)
     *se anulează tăietura
   altfel                                          [5.3.4.a]
     dacă lungimeFir=1 atunci
        *se taie bucata de lungime 1;
     □
     *afişare
  □
-------------------------------------------------------------
```

- În secvenţa [5.3.4.b] se prezintă o variantă de implementare în care:
  - 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.

```
-------------------------------------------------------------
/* Implementare a algoritmului de tăiere a firului */

int i,k,x;
char z[9];
void taie(int lungimefir)
  {
    if (lungimefir>1)
        {
          k=k+1;
          z[k-1]='.';
          taie(lungimefir-1);
          z[k-1]='_';
          taie(lungimefir-2);
          k=k-1;/*anulare taietura*/
        }
      else
        {                                          /*[5.3.4.b]*/
          printf("     ");
          for( i=1; i <=k; i ++)
             printf("%c", z[i-1]);
          if (lungimefir==1)
```

```
            printf(".");
          printf("\n");
        }
  }
```

- În figura 5.3.4.a apare rezultatul apelului procedurii `Taie` pentru `n = 4` şi `n = 5`.
    - Din această figură se poate deduce uşor maniera de execuție a procedurii (urma execuției) luând în considerare faptul că în reprezentarea grafică:
        - Caracterul '.' reprezintă apelul procedurii `Taie` pentru `n-1`,
        - Cracterul '_' reprezintă apelul procedurii `Taie` pentru `n - 2`.

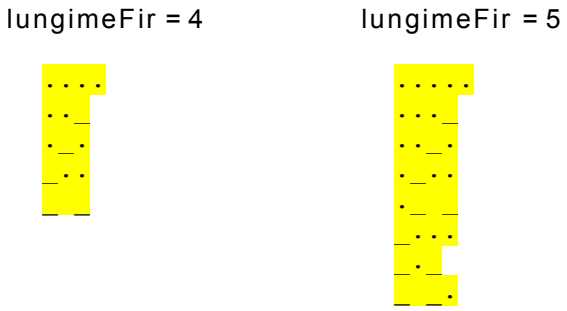


Fig.5.3.4.a. Execuția procedurii `Taie` pentru `n = 4` şi `n = 5`

- În figura5.3.4.b se prezintă **arborele de apeluri** al procedurii `Taie` pentru `n = 4` .
- Se observă că în fiecare nod al acestui arbore sunt precizate:
    - Succesiunea bucăților tăiate;
    - Apelul recursiv care se realizează;
    - Valoarea curentă a lui `k`;
    - În chenar îngroşat sunt încadrate secvențele care se afişează.

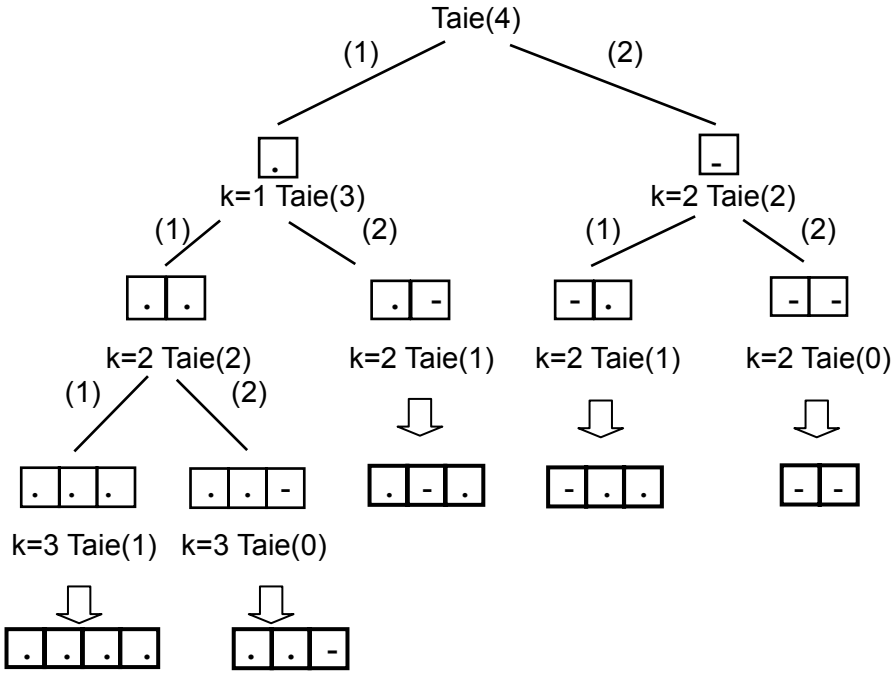


Fig.5.3.4.b. Arborele de apeluri al procedurii `Taie(4)`

- **Exemplul 5.3.4.b.** Algoritm pentru determinarea **tuturor** soluțiilor 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)`
  - În tablou, cu ajutorul caracterului '*' sunt reprezentați **pereții**,
  - Cu ajutorul caracterului ' ' sunt reprezentate **culoarele**.
  - Punctul de start este centrul labirintului,
- Drumul de **ieşire** se caută 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:
  - Dacă valoarea locului curent este ' '
    - 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).
  - Dacă valoarea locului curent **nu** este ' '
    - Se apelează recursiv procedura `Cauta` pentru cele patru puncte din vecinătatea imediată a locului curent, după direcțiile axelor de coordonate (dreapta, sus, stânga, jos).
- Pentru fiecare căutare 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 situații **revenindu-se** pe drumul parcurs până la proximul punct care permite selecția unui nou drum.
- **Ştergerea** se execută prin generarea unui caracter ' ' pe poziția curentă înainte de părăsirea procedurii.
  - Aceasta corespunde practic înfăşurării "firului parcurgerii" conform metodei "firului Ariadnei".
- În secvența [5.3.4.c] este prezentată schița de principiu a procedurii `Caută` iar în [5.3.4.d] o variantă de implementare `C`.

```
-------------------------------------------------------------
{Schita de principiu a algoritmului de căutare a drumului de
ieşire dintr-un labirint}

Procedura Cauta(x,y: coordonate);
  |dacă  *locul este liber atunci                [5.3.4.c]
  |  *marcheaza locul
  |  |dacă  *s-a ajuns la ieşire atunci *afişează drumul
  |  |  altfel
  |  |    Cauta(x+1,y); {dreapta} Cauta(x,y+1); {sus}
  |  |    Cauta(x-1,y); {stânga}  Cauta(x,y-1)  {jos}
  |  |
  |  *şterge marcajul
  |
-------------------------------------------------------------

/* Implementare a algoritmului de căutare a drumului de ieşire
dintr-un labirint */

void cauta(int x,int y)
  {
    if (m[x][y]==' ')
      {                                          /*[5.3.4.d]*/
        m[x][y]='#';
```

```
        if (((x % n)==0)||((y % n)==0)) ;
           /*afiseaza*/
          else
            {
              cauta(x+1,y);
              cauta(x,y+1);
              cauta(x-1,y);
              cauta(x,y-1);
            }
          m[x][y]=" ";
        }
    }
```

- 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.
- În figura 5.3.4.c se prezintă un exemplu de execuţie al acestei proceduri.

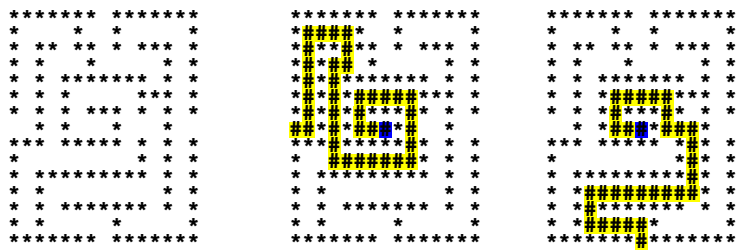


Fig.5.3.4.c. Exemplu de execuţie al procedurii `Cauta`

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 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 **exponențial**, iar efortul de căutare creşte în aceeaşi măsură.

- În mod obişnuit, arborii de căutare pot fi simplificați 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 regulile generale ale **euristicii.**

  - Mai degrabă:

    - **(1)** Se vor aborda **principiile** separării unei probleme în subproblemele care o rezolvă

    - **(2)** Se va utiliza **recursivitatea** în vederea soluționării acestora

5.4.1. Turneul calului

- **Specificarea** problemei:

  - 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 inițiale 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 execuția 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 Pascal}

PROCEDURE IncearcaMiscareaUrmatoare;
  BEGIN
    *iniţializează lista mişcărilor
    REPEAT
      *selectează posibilitatea următoare din lista
         mişcărilor
      IF *este acceptabilă THEN
        BEGIN                                        [5.4.1.a]
          *înregistrează mişcarea curentă;
          IF *tabela nu e plină THEN
              BEGIN
                IncearcaMiscareaUrmatoare;
                IF NOT *mişcare reusită THEN
                  *şterge înregistrarea curentă
              END
            ELSE
              *mişcare reuşită
```

```
        END
    UNTIL (*mişcare reuşită) OR (*nu mai există
                        posibilităţi în lista mişcărilor)
  END; {IncearcaMiscareaUrmatoare}
-------------------------------------------------------
/* Schiţa algoritmului pentru realizarea mişcării următoare a
calului - varianta pseudocod C */


void incearca_miscarea_urmatoare()
  {
    /*initializeaza lista miscarilor */
    do {
      /*selecteaza posibilitatea urmatoare din lista
         miscarilor*/
      if (/*este acceptabila*/)
        {                                          /*[5.4.1.a]*/
        /*înregistreaza miscarea curenta*/;
          if (/*tabela nu e plina*/)
              {
                incearca_miscarea_urmatoare();
                if (!/*miscare reusita*/)

                  /*sterge înregistrarea curenta*/;
              }
            else;
              /*miscare reusita*/
        }
    } while (!((/*miscare reusita*/) | (/*nu mai exista
posibilitati în lista miscarilor*/)));
  }
/*-------------------------------------------------------------
*/
```

- În continuare se vor preciza câteva dintre **structurile de date** care vor fi utilizate.

- În primul rând tabela de şah va fi reprezentată prin matricea `t` pentru care se introduce tipul `TipIndice` [5.4.1.b].

```
-------------------------------------------------------------
{Definirea structurilor de date: tabela de şah}

TYPE TipIndice = 1..n;                              [5.4.1.b]
     TipTabela = ARRAY[TipIndice,TipIndice] OF integer;
-------------------------------------------------------------
enum {n =10};
typedef unsigned tipindice;                    /*[5.4.1.b]*/
typedef int tiptabela[n][n];
/*-------------------------------------------------------*/
```

- După cum se observă, în câmpurile 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}


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

- În continuare se vor stabili **parametrii** specifici de apel ai procedurii care implementează algoritmul. Aceştia trebuie să precizeze:

- (1) **Condițiile de start** ale noii mișcări (parametri de intrare)
- (2) Să precizeze dacă mișcarea respectivă este **reuşită** sau **nu** (parametru de ieşire).

- Pentru prima cerință (1) sunt suficiente:
  - Coordonatele `x,y` de la care va porni mişcarea
  - Şi valoarea `i` care precizează numărul mutării curente necesară din rațiuni de înregistrare.

- Pentru cea de-a doua cerință (2) se introduce **parametrul de ieşire** `q` = **`true`** desemnând mişcare reuşită respectiv `q` = **`false`** nereuşită, din punctul de vedere al acceptării mişcării.

- Problema care se pune în continuare este aceea a **rafinării** propozițiilor precedate de caracterul "*****" în secvența [5.4.1.a].
  - În primul rând faptul că *****`tabela nu este plină` se exprimă prin $i < n^2$.
  - Se introduc variabilele locale `u` şi `v` pentru a preciza coordonatele unei **posibile destinații** a mișcării conform regulilor după care se efectuează saltul calului,
    - Propoziția *****`este acceptabilă` poate fi exprimată ca şi o combinație logică a condițiilor $1 \le u \le n$ **şi** $1 \le v \le n$ (adică noul câmp este pe tabelă) **şi** că el **nu** a fost vizitat anterior `(t[u,v] = 0)`.
  - Aserțiunea *****`înregistrează mişcarea` devine `t[u,v] = i;`
  - Aserțiunea *****`şterge înregistrarea curentă`, se exprimă prin `t[u,v] = 0`.
  - Se mai introduce variabila booleană locală `q1` utilizată ca şi parametru rezultat pentru apelul recursiv al procedurii. Variabila `q1` **substituie** de fapt condiția *****`mişcare reuşită` .
  - Se ajunge în definitiv la următoarea formulare a procedurii [5.4.1.d].

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

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

```
    q:= q1
  END; {Incearcă}
-------------------------------------------------------
/* Rafinarea procedurii Încearcă - pasul 1 de rafinare -
varianta C */

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

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

    /*initializeaza lista miscarilor */
    do {
      /*fie u,v coordonatele miscarii urmatoare conform
         regulilor sahului*/
      if ((1<=u<=n) && (1<=v<=n) && (t[u][v]==0))
        {
          t[u][v]=i;                          /*[5.4.1.d]*/
          if (i<n*n)
              {
                incearca(i+1,u,v,&q1);
                if (! q1)  t[u][v]=0;
              }
            else
              q1=true;
       }
    } while (!(q1 | (/*nu mai exista posibilitati în lista
                     miscarilor*/)));
    *q=q1;
  }    /*Incearca*/
/*-------------------------------------------------------------
*/
```

- Relativ la această secvență se fac următoarele precizări.

- Tehnica utilizată este cea cunoscută în literatura de specialitate sub denumirea de **tehnica "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` (următoarea din cele 8 posibile din lista de mişcări);

    - (3) Se încearcă realizarea mişcării următoare plecând de la poziția `u,v`.

    - (4) Dacă mişcarea **nu** este reuşită, respectiv s-au parcurs fără succes toate cele 8 posibilități ale listei de mişcări, se **anulează** mişcarea curentă (`u,v`), ea fiind lipsită de perspectivă (bucla **`REPEAT`**).

- Privind în perspectiva evoluției căutării, fiecare dintre cele 8 mişcări este tratată în manieră similară:

    - Respectiv 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 posibilităților

    - Dacă s-au epuizat toate posibilitățile, 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:

  - Este de ordinul 8 (din fiecare punct se pot selecta 8 posibilități de mişcare)

  - Are înălțimea n^2 (numărul de paşi necesari pentru soluția finală), element care explică **complexitatea** procesului de determinare a soluției problemei.

- În pasul următor şi ultimul de rafinare mai rămân câteva puncte de specificat.

- Precizarea saltului calului.

  - Fiind dată o poziție inițială <x,y> există opt **posibilități** pentru generarea coordonatelor destinației mişcării următoare <u,v>, care sunt numerotate de la 1 la 8 în fig.5.4.1.a.

	3		2	
4				1
		X		
5				8
	6		7	

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

  - O metodă simplă de obținere a lui u şi v din x şi y este de a aduna la acestea din urmă **diferențele specifice** de coordonate memorate în două tablouri, unul pentru x notat cu a şi unul pentru y notat cu b.

  - Indicele k precizează numărul **următorului candidat** din lista mişcărilor ($1 \le k \le 8$)

  - Formatul celor două tabele a şi b apare în continuare:

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

	3		2	
4				1
		X		
5				8
	6		7	

- Detaliile de implementare apar în programul [5.4.1.e], variantă Pascal respectiv C.

- Procedura recursivă este **inițiată** 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.

- În program această cerință a fost implementată cu ajutorul operatorului **IN**, utilizând mulțimea `S`, implementare care pentru valori mici ale lui `u` respectiv `v` este foarte eficientă.

```
{Determinarea Turneului Calului - varianta finală Pascal}

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

  PROCEDURE Incearca(i: integer; x,y: TipIndice;
                             VAR q: boolean);
    VAR k,u,v: integer;
        k: integer;
        q1: boolean;
    BEGIN
      k:= 0;
      REPEAT
        k:= k+1; q1:= false;
        u:= x+a[k]; v:= y+b[k];
        IF (1<=u<=n) AND (1<=v<=n) THEN
          IF t[u,v]=0 THEN
            BEGIN
              t[u,v]:= i;                        [5.4.1.e]
              IF i<n*n THEN
                  BEGIN
                    Incearca(i+1,u,v,q1);
                    IF NOT q1 THEN t[u,v]:= 0
                  END
                ELSE
                  q1:= true
            END
      UNTIL q1 OR (k=8);
      q:= q1
    END;{Incearca}
  BEGIN {programul principal}
    a[1]:= 2; b[1]:= 1;
    a[2]:= 1; b[2]:= 2;
    a[3]:=-1; b[3]:= 2;
    a[4]:=-2; b[4]:= 1;
    a[5]:=-2; b[5]:=-1;
    a[6]:=-1; b[6]:=-2;
    a[7]:= 1; b[7]:=-2;
    a[8]:= 2; b[8]:=-1;
    FOR i:=1 TO n DO
      FOR j:= 1 TO n DO t[i,j]:= 0;
    t[1,1]:= 1; Incearca(2,1,1,q);
    IF q THEN
        FOR i:= 1 TO n DO
          BEGIN
            FOR j:= 1 TO n DO Write('    ',t[i,j]);
            Writeln
          END
      ELSE Writeln('     nu exista solutie ')
  END.
```

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

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

enum {n =5};
```

```
typedef unsigned char tipindice;

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

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

void incearca(int i, tipindice x,tipindice y, boolean* q)
    {
     int k,u,v;
     boolean q1;
     k=0;
     do {
       k=k+1;
       q1=false;
       u=x+a[k-1]; v=y+b[k-1];
       if ((0<=u<=n-1) && (0<=v<=n-1))
          if (t[u-1][v-1]==0)
            {
              t[u-1][v-1]=i;                      /*[5.4.1.e]*/
              if (i<n*n)
                  {
                    incearca(i+1,u,v,&q1);
                    if (! q1)
                       t[u-1][v-1]=0;
                  }
                else
                  q1=true;
            }
      } while (!(q1 || (k==8)));
      *q=q1;
    }   /*Incearca*/

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


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

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

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

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

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

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

- **Caracteristica esențială** a acestui algoritm:
  - Înaintează spre soluția finală pas cu pas, **tatonând** și **înregistrând** drumul parcurs.
  - Dacă la un moment dat constată că drumul ales **nu** conduce la soluția dorită ci la o fundătură, **revine, ştergând** înregistrările paşilor până la proximul punct care permite o nouă alternativă de drum.
  - Această activitate se numeşte **revenire** (**backtraking**).

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

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

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

Procedura Incerca;
  BEGIN
    *iniţializează selecţia posibilităţilor;
    REPEAT
      *selectează posibilitatea următoare;
      IF acceptabilă THEN                          [5.4.2.a]
        BEGIN
          *înregistreaz-o ca şi curentă;
          IF *soluţie incompletă THEN
              BEGIN
                Incerca *pasul urmator;
                IF *nu este reuşit THEN
                  *şterge înregistrarea curentă
```

```
            END
          ELSE
            *pas reuşit (soluţie completă)
        END{IF}
    UNTIL (*pas reusit) OR (*nu mai sunt posibilităţi)
  END;{Incearca}
-------------------------------------------------------------
/* Model pricipial de rezolvare a unei probleme de tip (n,m).
Determinarea unei soluţii */

void Incerca()
  {
    *initializeaza selectia posibilitatilor;
    do
      *selecteaza posibilitatea urmatoare;
      if ( acceptabila )                          /*[5.4.2.a]*/
        {
          *înregistreaz-o ca si curenta;
          if ( *solutie incompleta )
              {
                Incerca *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 **`REPEAT`** [5.4.2.b].

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

Procedura Incerca1(i: TipPas);
  VAR posibilitate: TipPosibilitate;
  BEGIN
    posibilitate:= 0;{iniţializează selecţia
                      posibilităţilor}
    REPEAT
      posibilitate:= posibilitate+1;  {selecţie posibilitate
                                       următoare}
      IF *acceptabila THEN
        BEGIN                                    [5.4.2.b]
          *înregistreaz-o ca şi curentă;
          IF i<n THEN {soluţie incompletă}
              BEGIN
                Incerca1(i+1); {încerca pasul următor}
                IF *nereuşit THEN
                   *şterge înregistrarea curentă
              END
            ELSE
              *soluţie completă (afişare)
        END
    UNTIL *soluţie completă OR (posib=m)
  END;{Incerca1}
```

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


void Incerca1(TipPas i)
  {
    TipPosibilitate posib;
    posib=0;/*initializeaza selectia posibilitatilor*/
    do
      posib=posib+1;  /*selectie posib. urmatoare*/
      if ( *acceptabila )
        {                                    /*[5.4.2.b]*/
          *înregistreaz-o ca si curenta;
          if ( i<n ) /*solutie incompleta*/
              {
                Incerca1(i+1); /*încerca pasul urmator*/
                if ( *nereusit )
                   *sterge înregistrarea curenta
              }
            else
              *solutie completa (afisare)
        }
    while (!*solutie completa || (posib!=m))
  }/*Incerca1*/
/*------------------------------------------------------------*/
```

- (2) În cea de-a doua variantă, procedura `Incearca2` are drept parametru de apel o **posibilitate** de selecţie

    - Construcţia soluţiei se realizează apelând recursiv procedura pe rând 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}

Procedura Incearca2(posibilitate: TipPosibilitate);
  BEGIN
    IF *acceptabilă THEN
      BEGIN
        *înregistreaz-o ca şi curentă;
        IF *soluţie incompletă THEN
            BEGIN                                  [5.4.2.c]
              Incearca2(Posibilitate1);
              Incearca2(Posibilitate2);
                  ...
              Incearca2(Posibilitatem);
              *şterge înregistrarea curentă
            END
          ELSE
            *soluţie completă (afişare)
      END
  END;{Incearca2}
------------------------------------------------------------
/* Rezolvarea unei probleme de tip (n,m). Determinarea unei
soluţii - Varianta 2 */

void Incearca2(TipPosibilitate posib)
  {
    if ( *acceptabila )
      {
        *înregistreaz-o ca si curenta;
        if ( *solutie incompleta )
```

```
            {                                   /*[5.4.2.c]*/
              Incearca2(Posibilitate1);
              Incearca2(Posibilitate2);
                  ...
              Incearca2(Posibilitatem);
              *sterge înregistrarea curenta
            }
          else
            *solutie completa (afisare)
      }
  }/*Incearca2*/
/*-------------------------------------------------------*/
```

- Se presupune că la fiecare pas numărul posibilităților de examinat este **fix** (`m`) şi că procedura este apelată inițial prin `Incearca2(1)`.

- În continuare în cadrul acestui paragraf vor fi prezentate câteva aplicații ale algoritmilor cu revenire, care se pretează deosebit de bine unei abordări recursive.

5.4.3. Problema celor 8 regine

- Problema celor 8 regine reprezintă un exemplu bine cunoscut de utilizare a algoritmilor cu revenire.

- Această problemă a fost investigată de K.F. Gauss în 1850, care însă **nu** a rezolvat-o complet, întrucât până în prezent **nu** a fost găsită o soluție analitică completă.

    - În schimb, problema celor 8 regine poate fi rezolvată prin încercări, necesitând 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 amenințe pe celelalte.

- Se observă imediat că aceasta este o problemă de tip `(n,m)`:

    - Deoarece există 8 regine care trebuiesc plasate, deci soluția necesită 8 **paşi**,

    - Pentru fiecare din cele 8 regine existând, după cum se va vedea, 8 **posibilități** de a fi aşezate pe tabla de şah.

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

```
-------------------------------------------------------------
{Rezolvarea Problemei celor 8 regine - schița de principiu}

PROCEDURE Incerca(i: regina);
  BEGIN
    *inițializează selecția locului de plasare pentru
       a i-a regină
    REPEAT
      *selectează locul următor
      IF *loc sigur THEN
        BEGIN                                   [5.4.3.a]
          *plasează regina i
          IF i<8 THEN
              BEGIN
                Incearca(i+1);
```

```
              IF *încercare nereuşită THEN *ia regina
            END
          ELSE
            *încercare reuşită (i=8)
        END
    UNTIL *încercare reuşită OR (*nu mai exista locuri)
  END;{Incerca}
-------------------------------------------------------------
/* Rezolvarea Problemei celor 8 regine - schiţa de principiu */

void Incerca(regina i)
  {
    *initializeaza selectia locului de plasare pentru
       a i-a regina
    do
      *selecteaza locul urmator
      if ( *loc sigur )
        {                                   /*[5.4.3.a]*/
          *plaseaza regina i
          if ( i<8 )
              {
                Incearca(i+1);
                if ( *încercare nereusita ) *ia regina
              }
            else
              *încercare reusita (i=8)
        }
    while (!(*încercare reusita) || !(*nu mai exista locuri))
  }/*Incerca*/
/*-------------------------------------------------------------*/
```

- Sunt necesare câteva **precizări**.
    - Deoarece din regulile şahului se ştie că regina ameninţă **toate** câmpurile situate pe aceeaşi **coloană**, **rând** sau **diagonală** în raport cu câmpul pe care ea se află, **rezultă** că fiecare **coloană** a tablei de şah va putea conţine **o singură** regină.
    - Astfel 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 în 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 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.

- Soluția imediată este aceea a reprezentării tablei cu ajutorul unei **matrice** `t` de dimensiuni `8x8` , dar o astfel de reprezentare conduce la operații 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 informației, în cazul de față **nu** se vor reprezenta pozițiile reginelor pe tabla de şah ci faptul că o regină a fost sau nu plasată 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 9 regine - definirea structurilor de date}

VAR x: ARRAY[1..8] OF integer;
    a: ARRAY[1..8] OF boolean;
    b: ARRAY[b1..b2] OF boolean;                [5.4.3.b]
    c: ARRAY[c1..c2] OF boolean;
-------------------------------------------------------------
/*{Problema celor 9 regine - definirea structurilor de date */

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

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

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

- Se precizează că pe tabela de şah există 15 diagonale / (înclinate spre dreapta) şi 15 diagonale \ (înclinate spre stânga).

- Caracteristica unei diagonale / este aceea că **suma** coordonatelor `i` şi `j` pentru oricare câmp care îi aparține este o constantă,

- Pentru diagonalele \ este caracteristic faptul că **diferența** coordonatelor `i` şi `j` pentru oricare câmp este o constantă.

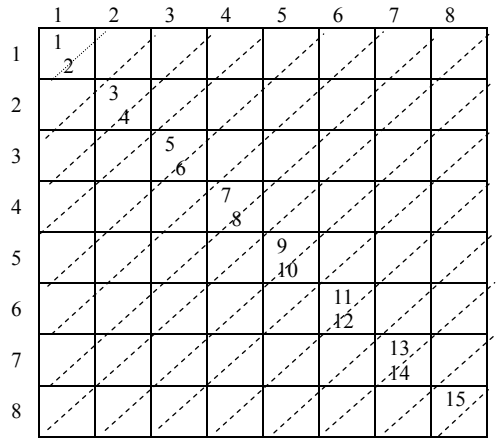
- În figura 5.4.3.a apar reprezentate aceste două tipuri de diagonale.

  - După cum se observă, pentru diagonalele / **sumele** `i+j` sunt cuprinse în domeniul `[2,16]`

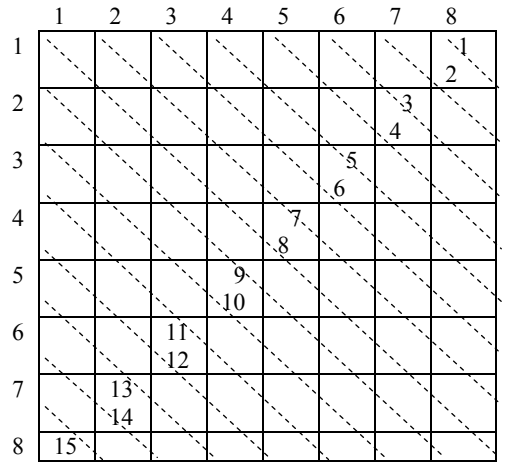
  - Iar pentru diagonalele \ **diferențele** aparțin domeniului `[-7,7]`.

- Aceste considerente fac posibilă alegerea valorilor limitelor `b1,b2,c1,c2` pentru indicii tabelelor `b` şi `c` [5.4.3.b].

- Una din posibilități este cea utilizată în continuare pentru care s-au ales `b1 = 2, b2 = 16, c1 = 0, c2 = 14`.



$2 \leq i + j \leq 16$



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

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

- Pentru `b1` şi `b2` s-au ales chiar limitele intervalului în care sumele indicilor iau valori,
- Intervalul în care iau valori **diferențele** indicilor a fost translatat cu valoarea 7 spre dreapta pentru a obține valori **pozitive** pentru indicele de acces în tabela `c`.
- Cu alte cuvinte, accesul în tabloul `b` destinat evidenței diagonalelor / se realizează prin `b[i+j]`
- Iar accesul în tabloul `c` destinat evidenței diagonalelor \ prin `c[i-j+7]`.
- Inițial, locațiile tablourilor `a,b` şi `c` se poziționează pe **`true.`**
- Cu ajutorul acestor reprezentări afirmația *`plasează regina` pe poziția `(i,j)`, `i` fiind coloana proprie, devine [5.4.3.d]:

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

```
 x[i]:= j; a[j]:= false; b[i+j]:= false;        [5.4.3.d]
 c[i-j+7]:= false
-----------------------------------------------------------
/*plaseaza regina i pe pozitia (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]
-----------------------------------------------------------
/*ia regina*/
                                      /*[5.4.3.e]*/
 a[j]=true; b[i+j]=true; c[i-j+7]=true
/*-----------------------------------------------------------*/
```

- 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] AND b[i+j] AND c[i-j+7]              [5.4.3.f]
-----------------------------------------------------------
/*sigura*/

   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].

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

PROGRAM Regine1;
  {găseşte o soluţie a problemei celor 8 regine}

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

PROCEDURE Incearca(i: integer; VAR q: boolean);
  VAR j: integer;
  BEGIN
    j:= 0;
    REPEAT
      j:= j+1; q:= false;
      IF a[j] AND b[i+j] AND c[i-j+7] THEN
        BEGIN
          x[i]:= j;                              [5.4.3.g]
          a[j]:= false; b[i+j]:= false;
          c[i-j+7]:= false;
          IF i<8 THEN
              BEGIN
                Incearca(i+1,q);
                IF NOT q THEN
                  BEGIN
                    a[j]:= true; b[i+j]:= true;
                    c[i-j+7]:= true
```

```
                END
              END
            ELSE
              q:= true
        END
    UNTIL q OR (j=8)
  END;{Incearca}
BEGIN {programul principal}
  FOR i:=1 TO  8 DO a[i]:= true;
  FOR i:=2 TO 16 DO b[i]:= true;
  FOR i:=0 TO 14 DO c[i]:= true;
  Incearca(1,q);
  IF q THEN
    FOR i:=1 TO 8 DO Write(x[i]);
  Writeln
END.
-------------------------------------------------------------
/*-----------------------------------------------------------*/
#include <stdio.h>
/*gaseste o solutie a problemei celor 8 regine*/

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

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

void incearca(int i, boolean* q)
  {
    int j;
    j=0;
    do {
      j=j+1; *q=false;
      if (a[j-1] && b[i+j-2] && c[i-j+7]){
          x[i-1]=j;                              /*[5.4.3.g]*/
          a[j-1]=false;
          b[i+j-2]=false;
          c[i-j+7]=false;
          if (i<8){
                incearca(i+1,q);
                if (!*q){
                    a[j-1]=true;
                    b[i+j-2]=true;
                    c[i-j+7]=true;
                  }
              }
            else
              *q=true;
        }
    } while (!(*q || (j==8)));
  }   /*Incearca*/
int main(int argc, const char* argv[])
{     /*programul principal*/
  for( i=1; i <= 8; i++)
    a[i-1]=true;
  for( i=2; i <=16; i++)
    b[i-2]=true;
  for( i=0; i <=14; i++)
    c[i]=true;
  incearca(1,&q);
  if (q)
    for( i=1; i <=8; i++)
        printf("%i ", x[i-1]);
  printf("\n");
```

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

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

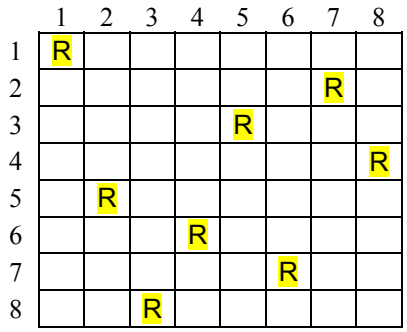


Fig.5.4.3.b. Soluția problemei celor 8 regine

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

- Modelul de determinare a unei soluții pentru o problemă de tip `(n,m)` poate fi uşor extins pentru a determina **toate** soluțiile unei astfel de probleme.

- Pentru aceasta este necesar ca generarea paşilor care construiesc soluția să se facă într-o manieră **ordonată** ceea ce garantează că un anumit pas **nu** poate fi generat decât o **singură** dată.

- Această proprietate corespunde căutării în arborele de apeluri, într-o manieră sistematică, astfel încât fiecare nod să fie vizitat o singură dată.

- De îndată ce o soluție a fost găsită şi înregistrată, se trece la determinarea soluției următoare pe baza unui proces de selecție sistematică până a la epuizarea **tuturor** posibilităților.

- Schema generală de principiu derivată din [5.4.2.a], care rezolvă această problemă apare în [5.4.4.a].

- În mod surprinzător, găsirea tuturor soluțiilor unei probleme de tip `(n,m)` presupune un algoritm mai **simplu** decât găsirea unei singure soluții.

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

Procedura Incearca;
  BEGIN
    FOR *toate posibilităţile de selecţie DO
      IF *selecţie acceptabilă THEN
        BEGIN
          *înregistreaz-o ca şi curentă        [5.4.4.a]
          IF *soluţia incompletă THEN
              Incearca *pasul urmator
            ELSE
              *evidenţiază soluţia;
```

```
          *şterge înregistrarea curentă
        END
  END;{Incearca}
-------------------------------------------------------------
/* Model pentru rezolvarea unei probleme de tip (n,m) -
determinarea tuturor soluţiilor */

void Incearca()
  {
    for ( *toate posibilitatile de selectie )
      if ( *selectie acceptabila )
        {
          *înregistreaz-o ca si curenta         /*[5.4.4.a]*/
          if ( *solutia incompleta )
              Incearca *pasul urmator
            else
              *evidentiaza solutia;
          *sterge înregistrarea curenta
        }
  }/*Incearca*/
/*-------------------------------------------------------*/
```

- Ca şi în cazul anterior, acest model principial va fi concretizat în două variante.

- (1) În prima variantă, procedura `Incearca1` are drept parametru de apel **numărul** pasului curent şi realizează explorarea posibilităților în bucla interioară **FOR** [5.4.4.b].

  - Deoarece pentru evidențierea tuturor posibilităților în fiecare pas trebuiesc parcurse toate valorile lui `k=[1,m]`, ciclul **REPEAT** a fost înlocuit cu unul **FOR**.

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

PROCEDURE Incearca1(i: TipPas);
  VAR posib: TipPosibilitate;
  BEGIN
    FOR posib:= 1 TO m DO
      IF acceptabilă THEN
        BEGIN
          *înregistreaz-o ca şi curentă;
          IF i<n THEN                                  [5.4.4.b]
              Incearca1(i+1)
            ELSE
              *afişează soluţia;
          *şterge înregistrarea
        END
  END;{Incearca1}
-------------------------------------------------------------
/* Rezolvarea unei probleme de tip (n,m). Determinarea tuturor
soluţiilor - Varianta 1 */

void Incearca1(TipPas i)
  {
   TipPosibilitate posib;
    for ( posib=1 pana_la m )
      if ( acceptabila ){
          *înregistreaz-o ca si curenta;
          if ( i<n )                               /*[5.4.4.b]*/
              Incearca1(i+1)
            else
              *afiseaza solutia;
          *sterge înregistrarea
        }
  }/*Incearca1*/
/*-------------------------------------------------------*/
```

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

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

Procedura Incearca2(posib: TipPosibilitate);
  BEGIN
    IF *acceptabilă THEN
      BEGIN
        *înregistreaz-o ca şi curentă;
        IF *soluţie incompletă THEN
            BEGIN                                  [5.4.4.c]
              Incearca2(Posibilitate1);
              Incearca2(Posibilitate2);
                      ...
              Incearca2(Posibilitatem);
            END
          ELSE
            *afişează soluţia
        *şterge înregistrarea curentă
      END
  END;{Incearca2}
------------------------------------------------------------
{Rezolvarea unei probleme de tip (n,m). Determinarea tuturor
soluţiilor - Varianta 2}

void Incearca2(TipPosibilitate posib)
  {
    if ( *acceptabila )
      {
        *înregistreaz-o ca si curenta;
        if ( *solutie incompleta )
            {                                /*[5.4.4.c]*/
              Incearca2(Posibilitate1);
              Incearca2(Posibilitate2);
                      ...
              Incearca2(Posibilitatem);
            }
          else
            *afiseaza solutia
        *sterge înregistrarea curenta
      }
  }/*Incearca2*/
/*-----------------------------------------------------------*/
```

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

```
------------------------------------------------------------
{Problema celor 8 regine - determinarea tuturor soluţiilor}

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

PROCEDURE Afisare;
  VAR k: integer;
  BEGIN
    FOR k:=1 TO 8  DO  Write(x[k]);            [5.4.4.d]
    Writeln
  END;{Afisare}
```

```
PROCEDURE Incearca(i: integer);
  VAR j: integer;
  BEGIN
    FOR j:=1 TO 8 DO
      IF a[j] AND b[i+j] AND c[i-j+7] THEN
        BEGIN
          x[i]:= j;
          a[j]:= false; b[i+j]:= false; c[i-j+7]:=
                                                  false;
          IF i<8 THEN
              Incearca(i+1)
            ELSE
              Afisare;
          a[j]:= true; b[i+j]:= true; c[i-j+7]:= true
        END{IF}
  END;{Incearca}

BEGIN {programul principal}
  FOR i:=1 TO  8 DO a[i]:= true;
  FOR i:=2 TO 16 DO b[i]:= true;
  FOR i:=0 TO 14 DO c[i]:= true;
  Incearca(1)
END.
-----------------------------------------------------------
/* Problema celor 8 regine - determinarea tuturor soluțiilor */

#include <stdio.h>

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

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

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

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

int main(int argc, const char* argv[])
{     /*programul principal*/
  for( i=1; i <= 8; i++) a[i-1]=true;
  for( i=2; i <=16; i++) b[i-2]=true;
  for( i=0; i <=14; i++) c[i]=true;
```

```
  incearca(1);
  return 0;
}
/*-------------------------------------------------------------
*/
```

- Algoritmul prezentat anterior determină cele 92 de soluții ale problemei celor 8 regine.

- De fapt, din cauza simetriei există doar 12 soluții distincte care apar evidențiate în figura 5.4.4.a.

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

Fig.5.4.4.a. Soluțiile problemei celor 8 regine

5.5. Structuri de date recursive

5.5.1. Structuri de date statice şi dinamice

- În cadrul capitolului 1 au fost prezentate structurile **fundamentale** de date: tabloul, articolul şi mulțimea.
- Ele se numesc **fundamentale** deoarece:
  - Constituie elementele de bază cu ajutorul cărora se pot forma 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:
  - (1) De a fixa **domeniul valorilor** pe care le pot lua aceste variabile,
  - (2) De a preciza **structura, dimensiunea** şi **amplasarea** zonelor de **memorie** care le sunt asociate.
- 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 informației mai complicate, a căror caracteristică principală este aceea că **se modifică structural** în timpul execuției programului,
  - Din acest motiv aceste structuri se numesc structuri **dinamice**.
- Este însă evident faptul, că la un anumit nivel de detaliere, componentele unor astfel de structuri sunt tipuri de date **fundamentale**.
- În general, indiferent de limbajul de programare utilizat, o **structură de date statică** este aceea care ocupă pe toată durata execuției programului căruia îi aparține, o zonă de memorie **fixă**, de volum **constant**.

- (1) Memoria necesară unei structuri statice se rezervă în faza de **compilare** deci înainte de lansarea programului.
- (2) Se consideră statice acele structuri care au **volumul** efectiv **constant**
- (3) 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 este funcție de maniera de **execuție** a algoritmului.
  - Cu alte cuvinte acest volum poate să crească sau să descrească în dependență de anumiți factori cunoscuți **numai** în timpul rulării.
  - În consecință alocarea memoriei la o structură dinamică are loc în timpul **execuției** programului.
- În general, orice **variabilă** declarată în partea de declarare a variabilelor, cu excepția celor de tip secvență, reprezintă o **structură statică**, pentru care se rezervă o zonă de memorie constantă.
  - Prin declararea unei variabile statice se precizează **numele** şi **tipul** ei
    - Numele este un identificator prin intermediul căruia programul respectiv programatorul poate face referire la această variabilă.
- Până în momentul de față, singurele structuri dinamice abordate au fost cele de tip **secvență**.
  - Datorită faptului că secvențele se presupun înregistrate în memorii externe, ele se consideră structuri dinamice **speciale** şi **nu** fac obiectul prezentului paragraf.
- De fapt atât structurile **dinamice** cât şi cele **statice** sunt formate în ultimă instanță 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 consecință **nu** se poate face referire la ele utilizând numele lor, deoarece practic **nu** au nume.
- **Referirea** unor astfel de structuri se face cu ajutorul unor **variabile statice**, definite special în acest scop 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 aplicații speciale impun definirea unui tip special de variabile numite variabile **indicator**.
- Valorile acestor variabile **nu** sunt date efective ci ele **precizează** (indică) locații de memorie care memorează date efective.
- Cu alte cuvinte, **valoarea** unei astfel de variabile indicator reprezintă o **referință** la o variabilă de un anumit tip precizat, numită **variabilă indicată**.

- Pentru a preciza natura unor variabile indicator, s-a introdus un nou tip de date abstract şi anume **tipul de date abstract indicator**.
- Descrierea de principiu a unui astfel de tip apare în [5.5.2.a].

TDA Indicator

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

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

Operatori:
      New(p) - procedură care alocă memorie pentru o
        variabilă indicată şi plasează valoarea adresei
        de memorie (indicatorul) în variabila p;
      p= Alloc(Type,dimension);
      Dispose(p) - procedură care eliberează zona de
        memorie (locația) corespunzătoare variabilei
        indicate p;
      Free(p);
      MemoreazăIndicator(p,q) - copiază indicatorul p în q;
      MemoreazăValoareIndicată(p,e) - copiază valoarea
        lui e în zona (locația) indicată de p;
      e:=FurnizeazăValoareIndicată(p) - funcție care
        returnează valoarea memorată în zona (locația)
        indicată de p;
      b:=IndicatorIdentic(p,q) - funcție booleană ce
        returnează valoarea true dacă p≡q, adică cele
        două variabile indicator indică aceeaşi locație
        de memorie.
-------------------------------------------------------------
```

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

5.5.2.2. Implementarea TDA Indicator cu ajutorul tipului pointer

- 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 **semnificația** variabilelor pointer.
    - **Valoarea** unei astfel de variabile este de fapt o **adresă de memorie** care poate fi adresa unei structuri dinamice sau a unei componente a ei.
    - În consecință, în limbaj de asamblare accesul la variabila indicată de o variabilă pointer se realizează prin **adresarea indirectă** a celei din urmă.
    - În limbajele de nivel superior acelaşi lucru se realizează prin ataşarea unor caractere speciale numelui variabilei pointer în dependență de limbajul utilizat [5.5.2.2.a].

```
-------------------------------------------------------------
//Precizarea unei variabile indicate
VariabilaIndicata ≡ *VariabilaPointer    [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 stabileşte 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 restricție măreşte **siguranța** 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 caracterul "*" în limbajul C [5.5.2.2.b].

```
-------------------------------------------------------------
//Definirea unui tip pointer - varianta C
tip_indicat  *variabila_pointer;    (C)      [5.5.2.2.b]
-------------------------------------------------------------
```

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

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

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

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

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

    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 (referință), **implementarea TDA Indicato**r este imediată şi realizată chiar la nivelul limbajului, în limbajele care **nu** dispun de această facilitate sau în situații speciale, implementarea acestui tip de date se poate realiza cu ajutorul **cursorilor**.
- Un **cursor** este o **variabilă întreagă** utilizată pentru a indica o **locație** într-un tablou de variabile indicate.
    - Ca 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 referință.
    - Interpretând valoarea unei variabile întregi ca şi un indice într-un tablou, în mod efectiv acea variabilă va **indica** locația respectivă din tablou.
- Cu ajutorul acestei tehnici, pot fi implementate practic toate structruile de date care presupun înlănțuiri, după cum se va vedea în capitolele următoare.
    - Este însă evident faptul că sarcina **gestionării** zonei care se alocă dinamic revine **exclusiv** programatorului, cu alte cuvinte utilizatorul trebuie să-şi dezvolte proprii operatori de tip `New - Dispose` respectiv `alloc – free`

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 care se apelează pe ele însele.
    - În cadrul acestui paragraf se va prezenta **extinderea** acestei proprietăți şi asupra **tipurilor de date** în forma aşa-numitelor "**structuri de date recursive**".
- Prin analogie cu algoritmii, prin **structură de date recursivă** se înțelege o structură care are cel puțin o **componentă** de acelaşi tip ca şi structura însăşi.
- Şi în acest caz, definițiile unor astfel de tipuri de date pot fi recursive în mod **direct** sau în mod **indirect** (vezi &5.1).
- Cel mai simplu **exemplu** de structură recursivă îl reprezintă **lista înlănțuită simplă** a cărei definire formală apare în [5.5.3.a].

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

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

- Un alt exemplu 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 părinților.
- Aceasta se exprimă formal astfel [5.5.3.b]:

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

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

- Datorită câmpurilor `urm` respectiv `tata` şi `mama`, care au acelaşi tip ca şi structura însăşi, tipurile definite sunt **recursive**.
- Se face precizarea că definițiile de mai sus **nu** sunt corecte, deoarece se utilizează identificatorii `TipLista` repectiv `TipGenealogie` înainte de a fi **complet** definiți (utilizarea unui tip în curs de definire).
    - Acest neajuns va fi remediat ulterior.
- Se constată uşor că cele două tipuri definesc structuri **infinite**.
    - Aceasta este o consecință directă a recursivității respectiv a faptului că există câmpuri de tip **identic** cu cel al structurii complete.
    - Este însă evident că în practică **nu** se pot prelucra structuri infinite.
- În realitate nici listele nici arborii genealogici nu sunt infiniți.
    - În cazul arborilor genealogici spre exemplu, urmărind suficient de departe ascendența oricărei persoane se ajunge la strămoşi despre care **lipseşte** informația.
    - Ținând cont de aceasta, se va modifica definiția tipului `TipGenealogie` astfel:
        - Dacă se ajunge la o persoană necunoscută atunci structura va conține un singur câmp notat cu 'XX',
        - În orice alt caz structura conține trei câmpuri conform definiției anterioare.
        - În figura 5.5.3.a se poate urmări o structură de `TipGenalogie` conform definiției 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, definiția tipului trebuie să conțină o **condiție**, de care depinde prezența efectivă a componentei (sau a componentelor) recursive.
    - Ca şi în exemplul de mai sus, în anumite condiții, componentele având acelaşi tip ca şi structura completă, pot să **lipsească**. În accepțiunea acestor condiționări pot exista structuri recursive finite.
- Structurile **recursive** pot fi implementate în limbajele de programare avansate numai în forma unor **structuri dinamice**.
- Prin **definiție**, orice componentă care are tipul identic cu structura completă, se **înlocuieşte** cu un **pointer** care indică acea componentă.
- În secvențele [5.5.3.c,d] apar pentru exemplificare definiri ale structurii recursive `TipGenealogie` în variantă Pascal respectiv C.

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

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

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

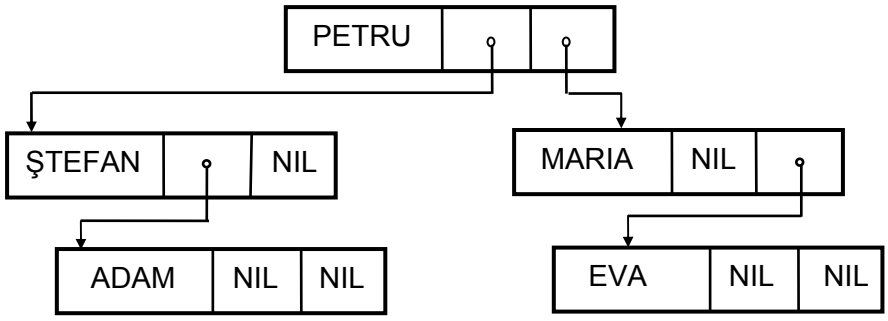


Fig.5.5.3.b. Implementarea unei structuri de date recursive

5.6. Rezumat

- O **definiție recursivă** se se referă la un obiect care se defineşte ca parte a propriei sale definiri.
- În programare **recursivitatea** este strâns legată de **iterație** şi se referă la auto apelarea unui subprogram de către el însuşi. Există **recursivitate directă** şi **indirectă**.
- **Implementarea recursivității** într-un limbaj de programare presupune un mecanism specializat de salvare în stivă a **contextului subprogramului** recursiv.
- Recursivitatea se utilizează cu predilecție la implementarea algoritmilor care se bazează pe definiții recursive: **calculul factorialului, numerele lui Fibonacci, algoritmul lui Euclid**.
- Orice algoritm recursiv se poate implementa şi în variantă iterativă. Există două cazuri denumite **recursivitate de sfârşit** care se poate transforma imediat în iterație (**vezi calculul**

factorialului) şi **recursivitate normală** care presupune rezolvarea de către programator a salvării contextelor într-o stivă anexă.

- Recursivitatea se poate utiliza cu mare succes la unele categorii de algoritmi cum ar fi **algoritmii de divizare (**exp. **Determinarea extremelor unui vector), algoritmii de reducere** *(*exp. **Determinarea permutărilor a n numere naturale)** şi **algoritmii care determină toate soluțiile de rezolvare ale unei probleme (**exp. **Secționarea firului).**
- Structurile de date pot fi **statice** sau dinamice. Pentru implementarea structurilor de date dinamice se definește **TDA indicator**. Cea mai cunoscută implementare a TDA indicator o reprezintă **pointerii.**
- O **structură de date recursivă** este o structură de date care are cel puțin o componentă de acelaşi tip cu structura însăşi.
- Structurile de date recursive se pot **implementa** numai în forma unor **structuri de date dinamice**. Astfel, prin definiție, **orice componentă care are tipul identic cu structura completă se înlocuieşte cu un pointer care indică acea componentă.**

5.7. Exerciții

1. Ce este o *definiție recursivă*? Dați *exemple* de definiții recursive.
2. Care este diferența dintre *iterație* şi *recursivitate* în activitatea de programare? Explicați *mecanismul de implementare al recursivității*.
3. Cum se poate *elimina recursivitatea*? Explicați *mecanismul eliminării recursivității* în cele două variante.
4. Implementați varianta recursivă şi varianta iterativă a *calculului factorialului* şi a *calculului şirului numerelor lui Fibonacci.* Testați funcționalitatea lor pentru diferite valori ale lui n.
5. Implementați varianta recursivă a *algoritmului lui Euclid*. Testați funcționalitatea lui pentru diferite valori ale lui m şi n.
6. Ce este un *algoritm de divizare*? Prezentați structura de principiu a unui astfel de algoritm. Cunoaşteți vreun algoritm care se încadrează în această categorie?
7. Care este principiul funcționării *algoritmilor de reducere*? Realizați o implementare a algoritmului de *determinare a tuturor permutărilor a n numere date*.
8. Care este principiul de funcționare al *algoritmului pentru determinarea tuturor soluțiilor unei probleme*. Extindeți problema tăierii firului la bucați de lungime 1,2,3 şi 4.
9. Redactați un program C care:
    - Citeşte de la tastatură elementele unei matrici care materializează un labirint
    - Caută şi *afişează toate drumurile de ieşire* din labirint

Obs. Programul va utiliza câte o procedură pentru citirea respectiv afişarea matricii labirint

10. Care este diferența dintre o *structură de date statică* şi o *structură de date dinamică*?
11. Definiți *TDA indicator*. Exemplificați implementarea lui în limbajul de programare C.
12. Ce este o *structură de date recursivă*. Precizați modalitatea de *definire a structurilor de date recursive*. Dați un exemplu.