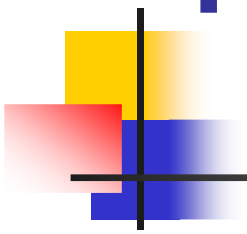
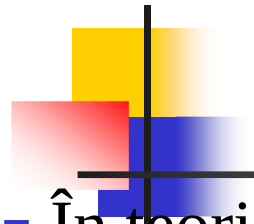
Metode generale de proiectare a algoritmilor şi programelor



- Metoda **Divide and Conquer**(Tehnica divizării)
- Metoda **Backtracking**(Algoritmi cu revenire)

Metode generale de proiectare

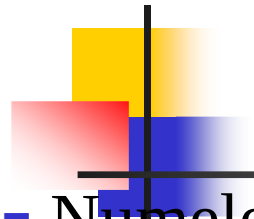
Introducere



- În teoria şi practica programării există un număr foarte mare de probleme pentru care trebuie găsite rezolvări. Din totalitatea acestor probleme se disting clase de probleme similare. Pentru problemele dintr-o asemenea clasă se poate aplica aceeaşi metodă generală de rezolvare, evident cu mici ajustări care depind de problema concretă.
- În timp s-au cristalizat mai multe metode generale de rezolvare a problemelor. Programatorii experimentați stăpânesc foarte bine aceste metode generale şi le aplică în mod automat atunci când au posibilitatea. Vom prezenta în continuare trei asemenea metode şi anume: ***Divide and Conquer, Greedy*** şi ***Backtracking***. Se recomandă studiul foarte atent al acestor metode, înțelegerea contextului în care ele se pot aplica şi însuşirea lor în asemenea mod încât aplicarea să poată deveni un automatism.

Metoda Divide and Conquer

Explicarea numelui



- Numele acestei metode de rezolvare a problemelor de programare provine din dictonul latin ***divide et impera***. Semnificația este următoarea: atunci când avem de rezolvat o problemă pe care, din diverse motive, o considerăm dificilă, o împărțim în mai multe subprobleme care să se poată rezolva mai uşor. După rezolvarea subproblemelor vom combina soluțiile lor pentru a obține soluția întregii probleme.
- Metoda de rezolvare *Divide and Conquer* se poate aplica la acele probleme care *se pot descompune în subprobleme de aceeaşi natură cu problema principală*, dar de dimensiuni mai mici.
- Se poate pune întrebarea : *Cum rezolvăm subproblemele?*. Răspunsul este: *În acelaşi mod în care am rezolvat problema principală*. => Metoda Divide and Conquer se pretează foarte bine la implementări recursive.

Metoda Divide and Conquer

Tehnica divizării - principii

- Este o metodă fundamentală de proiectare a algoritmilor care poate să conducă la soluții deosebit de eficiente.
- Principiul de bază al acestei tehnici este acela de a descompune în mod repetat o problemă complexă în două sau mai multe subprobleme de acelaşi tip, urmată de combinarea soluțiilor acestor subprobleme pentru a obține soluția problemei inițiale.
- Întrucât subproblemele rezultate din descompunere sunt de acelaşi tip cu problema inițială, metoda se exprimă în mod natural printr-o funcție recursivă.
- Apelul recursiv se continuă până în momentul în care subproblemele devin banale şi soluțiile lor evidente.

Metoda Divide and Conquer

Funcţia recursivă - pseudocod

```
void Divide(* parametri care definesc o problema) {
    if (* problema este una triviala) {
        * rezolva problema in mod direct;
        * returneaza rezultatele;}
    else{
        * imparte problema in subprobleme;
        for( fiecare subproblema )
            * apeleaza Divide(subproblema);
        * combina rezultatele subproblemelor;
        * returneaza rezultatele pentru problema;
    }
}
```

Metoda Divide and Conquer

Determinarea recursivă a elementelor min şi max

- Se dă un şir de *n* numere reale $\{x_0, x_1, ..., x_{n-1}\}$. Să se determine valoarea *minimă* şi valoarea *maximă* din acest şir de numere.
- Putem aplica tehnica *Divide and Conquer*, împărțind şirul de numere în două părți. Determinăm minimul şi maximul pentru fiecare din cele două părți, iar pe urmă determinăm maximul global prin compararea celor două maxime parțiale şi minimul global prin compararea celor două minime parțiale.
- Pentru implementare vom defini o funcție recursivă care va căuta minimul şi maximul într-o secvență a şirului. Inițial vom apela această funcție pentru întregul şir. Funcția se va apela pe ea însăşi recursiv, pentru jumătatea stângă şi pentru jumătatea dreaptă a secvenței .

Metoda Divide and Conquer

Determinarea recursivă a elementelor min şi max

```
#include <stdio.h>
/* Declaram sirul de numere direct in cod. Alternativ, el
   poate fi citit de la tastatura sau din fisier. */
#define N 10
int x[ ] = {10, 5, 23, -11, 4, 2, 0, -6, 66, 40};
int comp = 0; /*Numara cate comparatii se fac in total. */
/* Functie care determina minimul si maximul dintr-o
   secventa a sirului de numere. Secventa este
   delimitata de indicii "st" si "dr". Valorile minima si
   maxima gasite vor fi returnate prin pointerii "min" si
   respectiv "max" primiti ca si parametri. */
```

Metoda Divide and Conquer

Determinarea recursivă a elementelor min şi max

```
void minmax(int st, int dr, int *min, int *max) {
   int mijloc, min_st, max_st, min_dr, max_dr;
   printf("Caut in secventa [%d..%d].\n", st, dr);
   if (st == dr) {
     /* Daca secventa contine un singur numar, atunci el
     este atat minim cat si maxim. */
         *min = x[ st ];
         *max = x[ st ]; }
  else if (st == dr - 1) {
   /*Daca secventa contine doua numere, atunci  facem
     o comparatie pentru a gasi minimul si maximul. */
```

Metoda Divide and Conquer

Determinarea recursivă a elementelor min şi max

```
        comp++;
        if (x[ st ] < x[ dr ]){
            *min = x[ st ];     *max = x[ dr ]; }
        else {
            *min = x[ dr ];    *max = x[ st ]; }     }
    else  {
        /* Daca avem mai multe numere, atunci divizam
        problema in subprobleme. */
      mijloc = (st + dr) / 2;
      minmax(st, mijloc, &min_st, &max_st);
      minmax(mijloc+1, dr, &min_dr, &max_dr);
```

Metoda Divide and Conquer

Determinarea recursivă a elementelor min şi max

```
/* Combinarea rezultatelor partiale. Comparam
 minimele partiale si maximele partiale intre ele. */
comp++;
if (min_st < min_dr)
   *min = min_st;
else
   *min = min_dr;
comp++;
if (max_st > max_dr)
   *max = max_st;
else
   *max = max  dr;  }  }
```

Metoda Divide and Conquer

Determinarea recursivă a elementelor min şi max

```
int main(void) {
    int i, min, max;
    printf("Avem %d numere.\n", N);
    for (i=0; i<N; i++)   /* Afisam sirul de numere. */
        printf("%d ", x[ i ]);
    printf("\n\n");
    minmax(0, N-1, &min, &max);/* Apelam functia recursiva.*/
    printf("\nMinimul este %d.\n", min);
    printf("Maximul este %d.\n", max);
    printf("Comparatii facute: %d.\n", comp);
    return 0; }
```

Metoda Divide and Conquer

Problema Turnurilor din Hanoi

- Se dau trei tije notate cu *a*, *b*, *c*. Pe tija *a* sunt ordonate discuri în ordine crescătoare. Să se realizeze programul C care mută toate cele *n* discuri de pe tija *a* pe tija *b* folosind tija *c* ca auxiliară, astfel încât, în final, discurile să fie ordonate tot crescător. În timpul operațiilor de mutare este interzisă plasarea unui disc mai mare peste unul mai mic.
- Problema se poate codifica prin următorul cvartet : (*n*,*a*,*b*,*c*). Dacă am găsi o modalitate de a muta *n-1* discuri de pe tija *a* pe tija intermediară *c*, atunci am putea să mutăm discul cel mai mare de pe tija *a* pe tija *b*. Pe urmă ar trebui să aducem cele *n-1* discuri de pe tija *c* pe tija *b* şi problema ar fi rezolvată. Pentru a muta *n-1* discuri de pe tija *a* pe tija *c*, putem folosi ca tijă intermediară tija *b*. La fel, pentru a muta înapoi cele *n-1* discuri de pe tija *c* pe tija *b*, putem folosi ca tijă intermediară, tija *a*.
- Putem reformula cele de mai sus în felul următor: problema (*n*,*a*,*b*,*c*) se rezumă la problema (*n-1*,*a*,*c*,*b*), urmată de mutarea discului de diametru maxim de pe *a* pe *b*, urmată de problema (*n-1*,*c*,*b*,*a*).

Metoda Divide and Conquer

Problema Turnurilor din Hanoi

```
void hanoi (int n, char t_initial, char t_final,  char
    t_intermediar) {
   if (n > 1) {
        /* Daca avem mai mult de un disc de mutat,  atunci
        descompunem problema in  subprobleme. */
        hanoi(n-1, t_initial, t_intermediar, t_final);
        printf("%c -> %c\n", t_initial, t_final);
        hanoi(n-1, t_intermediar, t_final, t_initial); }
   else {
        /* Daca avem un singur disc de mutat, atunci il mutam
        direct. La acest nivel problema are o rezolvare triviala*/
        printf("%c -> %c\n", t  initial, t  final); }}
```

Metoda Backtracking

Algoritmi cu revenire – explicarea numelui

- Pentru a înțelege semnificația cuvântului *backtracking* vom reda definiția din *Cambridge Online Dictionary*, http://dictionary.cambridge.org/ :

to go back along a path you have just followed

- Ideea de bază este aceea de *revenire pe calea parcursă*. Algoritmii de tip backtracking încep să exploreze spațiul soluțiilor în mod exhaustiv, pe toate căile posibile. Atunci când pe calea curentă de explorare se constată că nu mai sunt şanse să se ajungă la o soluție validă, se revine cu un pas înapoi şi se abordează o altă cale de explorare.

- În concluzie, metoda Backtracking constă în efectuarea unor **încercări repetate**, în vederea găsirii soluțiilor, **cu posibilitatea revenirii** în caz de eşec.

Metoda Backtracking

Algoritmi cu revenire - principii

- Soluția se poate reprezenta sub forma unui vector $X=(x_0,x_1, \ldots, x_{n-1})$, $X \in S = S_0 \times S_1 \times \ldots \times S_{n-1}$, unde mulțimile $S_0, \ldots, S_{n-1}$ sunt mulțimi finite. Pentru fiecare problemă concretă sunt date anumite relații între componentele $x_0,x_1, \ldots, x_{n-1}$ ale vectorului X, numite **condiții interne**. Mulțimea S reprezintă **spațiul soluțiilor posibile**. Soluțiile posibile care satisfac condițiile interne se numesc **soluții rezultat**.

În continuare, exprimăm condițiile care trebuie satisfăcute sub forma unei funcții logice notată: $Solutie(x_0,x_1,\ldots,x_{n-1})$. Un element $X=(x_0, x_1, \ldots, x_{n-1}) \in S$ este soluție a problemei dacă funcția *Solutie* aplicată componentelor lui X va returna valoarea *true*.

Metoda Backtracking

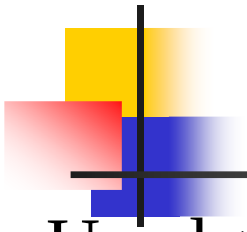
Algoritmi cu revenire - principii

- Scopul algoritmului concret poate să fie determinarea unei soluții rezultat sau a tuturor soluțiilor rezultat, fie în scopul afişării lor, fie pentru a alege una optimă din punctul de vedere al unor criterii de optimizare (minimizare sau maximizare).

- O metodă simplă de selectare a soluțiilor rezultat este aceea **de a** *genera toate soluțiile posibile şi de a verifica satisfacerea condițiilor interne (căutare exhaustivă* în întreg spațiul soluțiilor posibile*).* Această metodă necesită însă un timp de execuție foarte mare şi nu se aplică decât rar în practică.

Metoda Backtracking

Algoritmi cu revenire - principii



- Un algoritm backtracking performant, ca şi în cazul Greedy de altfel, evită generarea tuturor soluțiilor posibile. În acest scop, elementele vectorului X primesc, pe rând şi în ordine, valori. După asocierea unei valori lui x_k se verifică îndeplinirea unor **condiții de continuare** pentru secvența $(x_0, \ldots, x_k)$ şi numai apoi se trece la încercarea de asociere a unei valori pentru x_{k+1}: funcția *Continuare*$(x_0, x_1, ..., x_k)$.
- Neîndeplinirea acestor condiții ne arată, încă din această fază, că soluția finală nu poate fi o soluție rezultat. În cazul unui eşec la această verificare, se alege o altă valoare pentru $x_k \in S_k$ sau, dacă S_k a fost epuizat, se micşorează k cu o unitate şi procesul de alegere se repetă.

Metoda Backtracking

Procedura generală - pseudocod

```
k = 0;
while (k >= 0){
    do {
        * alege urmatorul x[k] din multimea S[k];
        * evalueaza Continuare(x[0], x[1], ..., x[k]); }
    while ( !Continuare(x[0], x[1], ..., x[k]) &&
            (* mai sunt elemente de ales din multimea S[k]) );
    if (Continuare(x[0], x[1], ..., x[k])) {
        if (k == n-1) {
            if (Solutie(x[0], x[1], ..., x[n-1])) * afiseaza solutie;  }
        else {  k = k + 1; } }
    else { k = k – 1; } }
```

Metoda Backtracking

Algoritmi cu revenire - caracteristici

În concluzie:

- Numele metodei (algoritmi cu revenire) provine de la **revenirile** care se efectuează în caz de eşec.

- Condițiile de *continuare* derivă din condițiile *interne* ale problemei.

- Alegerea optimă a condițiilor de continuare poate determina reducerea numărului de calcule (încercări) care urmează să fie efectuate (viteza programului).

- Acest proces de încercări repetate şi revenire în caz de eşec (cu reluarea altei valori şi continuare) se exprimă în mod natural şi în manieră recursivă.

Metoda Backtracking

Soluţia recursivă

- Se consideră că numărul elementelor care se pot examina la fiecare apel (cardinalitatea mulţimilor S_k) în vederea adăugării lor la soluție, este fix (m), iar numărul componentelor soluției este de asemenea fix (n) :

```
void Incearca (int k) {
  int i;
  for(i=0; i<m; i++) {
    * selectează elementul nr. i;
    if (* este acceptabil) {
      * înregistreaza elementul selectat;
      if (k<n-1)
         Incearca (k+1);
      else   * tipareşte soluție;
      * şterge înregistrarea;} } }
```

Metoda Backtracking

Problema celor 8 regine (Eight Queens)

- Se cere să se realizeze programul care să plaseze opt regine pe o tablă de şah, astfel încât nici una dintre ele să nu le ameninţe pe celelalte. La jocul de şah, o regină "ameninţă" pe linii, coloane şi diagonale, pe orice distanţă.

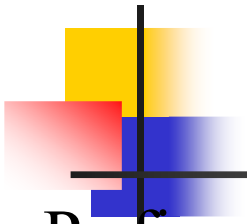
Această problemă a fost investigată de Carl Friedrich Gauss în 1850 (care însă nu a rezolvat-o complet). Nici până în prezent problema nu are o soluţie analitică satisfăcătoare. În schimb ea poate fi rezolvată prin încercări, necesitând o mare cantitate de muncă, răbdare şi acurateţe (condiţii în care calculatorul se descurcă excelent).

Problema are 92 de soluţii din care, din motive de simetrie, doar 12 sunt diferite.

Problema poate fi uşor extinsă pentru *n* regine plasate pe o tablă pătrată cu *n* linii şi *n* coloane.

Metoda Backtracking

8 regine – rezolvare



- Pe fiecare linie sau coloană de pe tablă se va afla o singură regină. Se va parcurge tabla de şah linie cu linie (k=0..7), iar în cadrul unei linii coloană cu coloană (i=0..7) şi se vor plasa reginele în acele pătrate care nu sunt în “priza reginelor” plasate anterior. Pentru parcurgerea tablei se va utiliza tehnica backtracking.
- Deoarece pe fiecare linie a tablei de şah se poate găsi exact o regină, o soluție rezultat se poate reprezenta sub forma unui vector $C = (c_0,..., c_7)$ unde c_k reprezintă coloana pe care se află regina de pe linia k ($c_k \in \{0, \ldots,7\}$).
- *Spațiul soluțiilor posibile* este produsul cartezian

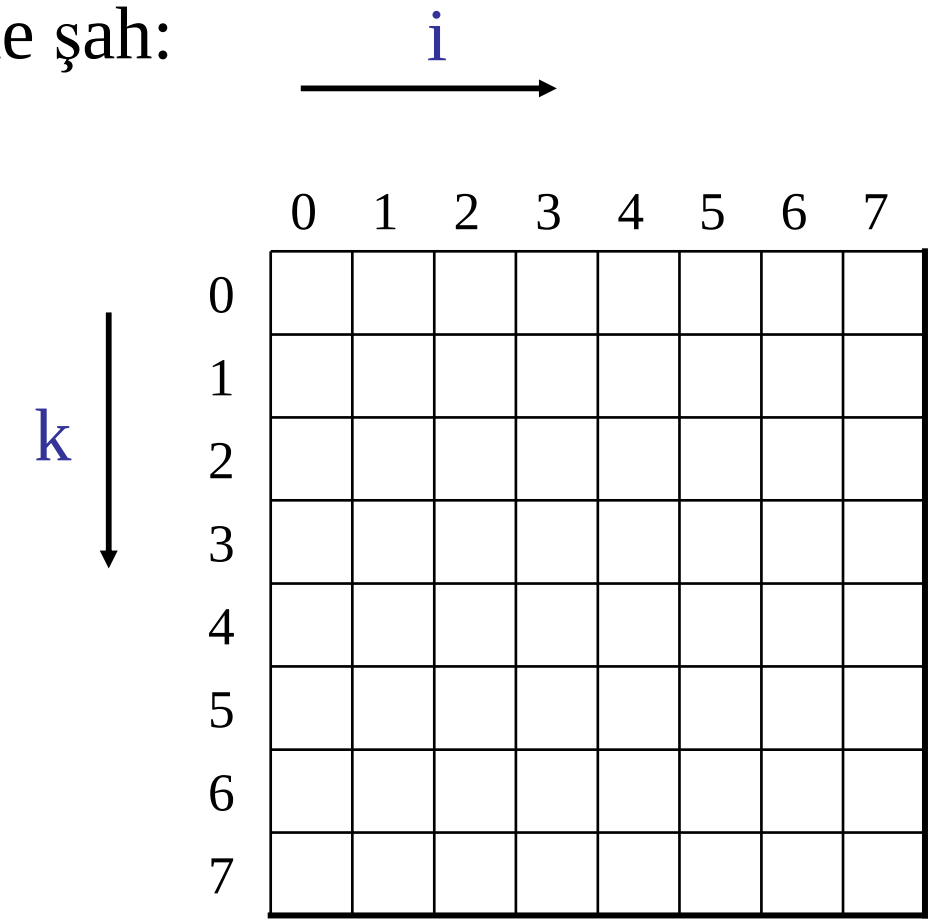
$$S = C \times C \times C \times C \times C \times C \times C \times C$$

- *Condițiile interne*, rezultă din regulile şahului şi sunt reprezentate de faptul că două dame nu se pot afla pe o aceeaşi coloană sau o aceeaşi diagonală.

Metoda Backtracking

8 regine – organizarea tablei de şah

Tabla de şah:



Metoda Backtracking

8 regine – condiţiile interne

Funcţia *Solutie* trebuie să verifice dacă nu există regine care se află pe aceeaşi coloană sau dacă nu cumva există regine care se atacă pe diagonală. Verificarea este simplă. Trebuie să verificăm că între elementele (c_0, c_1, c_2, c_3, c_4, c_5, c_6, c_7) nu există două care au aceeaşi valoare. Aceasta ar însemna că avem două regine pe aceeaşi coloană. Apoi mai trebuie să verificăm că $\forall i,k \in \{0, 1, 2, 3, 4, 5, 6, 7\}$, $|i-k| \neq |c_i-c_k|$. Aceasta este condiţia ca să nu existe două regine care se atacă pe diagonală. Verificări similare vor fi efectuate pe parcurs de funcţia *Continuare.*

8 Regine
Implementarea programului

```
#include <stdio.h>
#include <stdlib.h>
#define N 8
#define INVALID -1
int main(void) {
int c[N];
   int k, i;
   int continuare, ataca;
   int count = 0;  // Numaram cate solutii sunt gasite
   for (i=0; i<N; i++)
      c[i] = INVALID; //Initializam toate elem. din vectorul c
```

8 Regine
Implementarea programului

```
k = 0;
while (k >= 0) {
      /*  Ne vom opri atunci cand  am epuizat elem. din
      multimea  C[k] sau atunci cand intalnim un element
      pentru care functia Continuare returneaza true. */
    do {
     if (c[k] == INVALID)
        // nu s-a incercat plasarea reginei  de pe linia k
        c[k] = 0; // plasam regina de pe linia "k" pe coloana 0
    else //   incercam sa plasam regina pe urmatoarea col.
      c[k]++;
        /* Daca nu am depasit numarul de coloane de pe
linie, atunci  trecem la evaluarea functiei Continuare. */
```

8 Regine
Implementarea programului

```
if (c[ k ] < N) {
   /* Ne asiguram ca regina pe care vrem sa o plasam pe
      linia "k" si coloana "c[k]" nu ataca nici una din
       reginele deja plasate*/
   ataca = 0;
   for (i=0; !ataca && (i<k); i++) {
/* Verificam daca mai exista o regina pe aceeasi coloana*/
         if (c[ i ] == c[ k ])
               ataca = 1;
/*Verificam daca noua regina ataca alta, pe diagonala*/
          else if (abs(i-k) == abs(c[ i ]-c[ k ]))
               ataca = 1;   }
```

8 Regine

Implementarea programului

```
        /* Daca noua regina nu ataca nici una din reginele
           plasate anterior, atunci  functia Continuare
           returneaza true(1), altfel returneaza false (0) */
              if (!ataca)
                 continuare = 1;
              else
                 continuare = 0;
    }
   /* Daca s-a depasit numarul coloanelor de pe  o linie,
      functia Continuare returneaza  automat false. */
          else
             continuare = 0;  }
while (!continuare && (c[k] < N));
```

8 Regine

Implementarea programului

```
/* Daca am obtinut true (1), din functia Continuare, atunci
consideram regina plasata si continuam algoritmul */
        if (continuare) {
            /* Daca am plasat toate N reginele, atunci afisam
             solutia gasita. */
            if (k == N-1) {
                for (i=0; i<N; i++)
                    printf("%d ", c[ i ]);
                printf("\n");
                count++;  }
            else  k++; }
```

8 Regine

Implementarea programului

```
/* S-au epuizat toate variantele de  plasare a reginei de
pe linia k. Marcam cu INVALID  elementul c[ k ] si
revenim cu  un pas inapoi pe calea de cautare, la regina
k-1. */
          else {
              c[ k ] = INVALID;
              k--; }
    }
     printf("%d solutii\n", count);
     return 0;
}
```

Metoda Backtracking

8 regine – algoritmul recursiv

```
void Incearca(int k){
int i= -1;
  do {
    i++;
    Succes =regina poate fi pusa in pozitia (k,i)
    if (Succes){
          Ocupa;      /*pune regina*/
          C[k] = i;   /*memoreaza coloana*/
          if (k<7)
            Incearca(k+1);
          else
            Tipareste;
          Elibereaza; /*ia regina*/  }}
  while (i<7);  }
```

Metoda Backtracking

Circuitului calului (Knight's Tour)

- Fiind dată o tablă pătrată cu $n \times n$ elemente, se cere să se realizeze programul C pentru găsirea traseului care trece prin toate elementele tablei, începând cu cel de coordonate *(i,j)*, utilizând săritura calului dată de regulile şahului. Prin fiecare element se va trece o singură dată.

Operația esențială pe care o are de rezolvat algoritmul este executarea unei mişcări următoare sau constatarea că nu mai este posibilă o astfel de mişcare şi revenirea la mişcarea anterioară.

Caracteristica esențială a acestui algoritm este aceea că înaintează spre soluția finală pas cu pas, încercând şi înregistrând drumul parcurs. Dacă la un moment dat se observă că drumul ales nu conduce la soluția dorită şi se blochează, se revine ştergând înregistrările paşilor până la proximul punct care permite o nouă alternativă de drum.

Metoda Backtracking
Circuitului calului – structurile de date

- Pentru o tablă de dimensiune *N* vom memora soluțiile într-un vector, *c*, de lungime *N*N*.

- Fiecare element din vector va fi la rândul lui un vector cu două elemente, primul element va memora linia de pe tablă, iar al doilea element va memora coloana de pe tablă, corespunzătoare poziției respective:

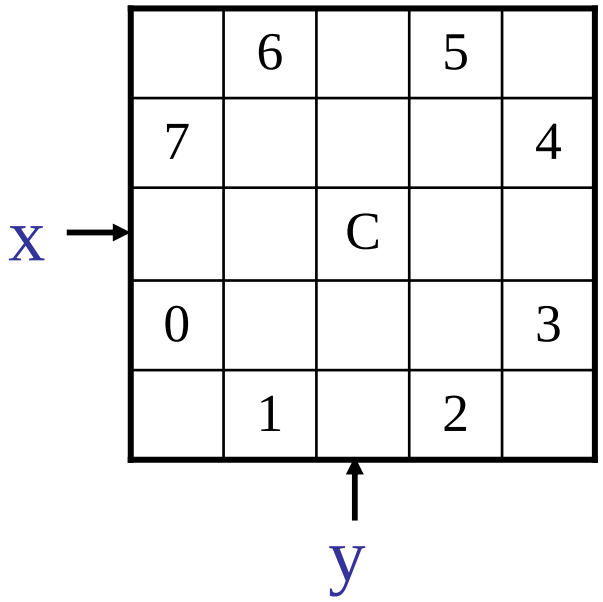
  *int c [N*N][2];*

- *count* – numărul de soluții găsite

Metoda Backtracking

Circuitului calului – săritura

Noile coordonate, pentru o săritură, se calculează din cele curente adunând nişte valori fixe ±1, ±2, înregistrate în tablourile *dx* şi *dy*, şi care definesc cele opt mişcări viitoare *(k=0..7)* posibile dintr-un punct oarecare *(x,y)* :



u = x + dx [k]

v = y + dy [k]

Metoda Backtracking

Circuitului calului – programul

```
#include <stdio.h>


#define N 5
int dx [8] = {-1, -2, -2, -1,  1,  2,  2,  1};
int dy [8] = {-2, -1,  1,  2,  2,  1, -1, -2};
int c[N*N][2];
int count = 0;
```

Metoda Backtracking
Circuitului calului – programul

```
void Incearca(int pas) {
    int i, j, continuare;

    if (pas == N*N)  {
       for (i=0; i<pas; i++)
           printf("(%d,%d) ", c[i][0], c[i][1]);
       printf("\n");
       count++;   }
```

Metoda Backtracking

Circuitului calului – programul

```
else
    for (i=0; i<8; i++)  {
        c[pas][0] = c[pas-1][0] + dy [i];
        c[pas][1] = c[pas-1][1] + dx [i];
        if ((c[pas][0]>=0) && (c[pas][0]<N) &&
                    (c[pas][1]>=0) && (c[pas][1]<N))  {
            continuare = 1;
            for (j=0; continuare && (j<pas); j++)
                if ((c[j][0]== c[pas][0]) && (c[j][1] == c[pas][1]))
                    continuare = 0;
            if (continuare)  Incearca(pas+1);  }  }  }
```

Metoda Backtracking

Circuitului calului – programul

```
int main(void){
    int i,j;
    for (i=0; i<N; i++)
        for (j=0; j<N; j++)
        {
            c[0][0] = i;
            c[0][1] = j;
            Incearca(1);
        }
    printf("%d solutii\n", count);
    return 0;
}
```