**Facultatea de Matematica si Informatica**
**Universitatea din Bucuresti**

# Tablouri

## Operatii pe tablouri bidimensionale

**Lectii de pregatire pentru Admitere**

**07 / 03 / 2020**

# Operatii pe tablouri bidimensionale

## Cuprins

# 0. Tablouri unidimensionale – scurta recapitulare

| C / C++ | Declarare | Pascal |
|---|---|---|
| int a[20];<br>double b[30];<br>char c[23]; | | var a : array [1..20] of integer;<br>var b : array [1..30] of double;<br>var c : array [1..23] of char; |

double tab [100] ;

| 0.3 | -1.2 | 10 | 5.7 | ... | 0.2 | -1.5 | 1 |
|---|---|---|---|---|---|---|---|
| **0** | **1** | **2** | **3** | | **97** | **98** | **99** |

tab [3] = 5.7;

int a[5];

| 3 | -12 | 10 | 7 | 1 |
|---|---|---|---|---|
| **0** | **1** | **2** | **3** | **4** |

a [1] = -12;

char b1 [34];

| A | & | * | + | ... | c | M | # |
|---|---|---|---|---|---|---|---|
| **0** | **1** | **2** | **3** | **...** | | | |

b1 [1] = '&';

# 0. Tablouri unidimensionale – scurta recapitulare

## Varianta C / C++

**Cantitatea de memorie necesara pentru inregistrarea unui tablou este direct proportionala cu tipul si marimea sa.**

tip nume [dimensiune] → **sizeof(nume) = sizeof (tip) * dimensiune;**

```
double tab [100] ;

int a[5];

char b1 [34];

printf("Stocarea unui tablou de elemente double = %d octeti\n",sizeof(tab));
printf("Stocarea unui tablou de elemente int = %d octeti\n",sizeof(a));
printf("Stocarea unui tablou de elemente char = %d octeti\n",sizeof(b1));
```

```
C:\Users\Ank\Desktop\testSizeof\bin\Debug\testSizeof.exe
Stocarea unui tablou de elemente double = 800 octeti
Stocarea unui tablou de elemente int = 20 octeti
Stocarea unui tablou de elemente char = 34 octeti
```

# 0. Tablouri unidimensionale – scurta recapitulare

**C / C++**

**Pascal**

**Traversare** ( complexitate **O(n)** )

```
for (i = 0; i<n; i++)
// viziteaza a[i];
```

```
for i:= 1 to n do
 { viziteaza a[i];}
```

**Cautare** ( **liniara** – complexitate **O(n)** )

```
int t = 0;
for (i = 0; i<n; i++)
      if (a[i]==x) t = 1;
if (t==0) // cautare fara succes
```

```
var t : boolean;
t := false;
for i:= 1 to n do
    if (a[i] = x) then t := true;
if (t = false) then
      {cautare fara succes};
```

# 0. Tablouri unidimensionale – scurta recapitulare

**C / C++**

**Pascal**

**Inserare** (valoare **val** pe pozitia **poz**)

```
for (i = n-1; i >= poz; i--)
      a[i+1] = a[i];
a[poz] = val;
n++;
```

```
for i:= n downto poz do
     a[i+1] := a[i];
a[poz]:=val;
n := n+1;
```

**Stergere** (valoare de pe pozitia **poz**)

```
for (i = poz; i<n-1; i++)
      a[i] = a[i+1];
n--;
```

```
for i := poz to n-1 do
     a[i] := a[i+1];
n:=n-1;
```

# 1. Tablouri bidimensionale – notiuni teoretice

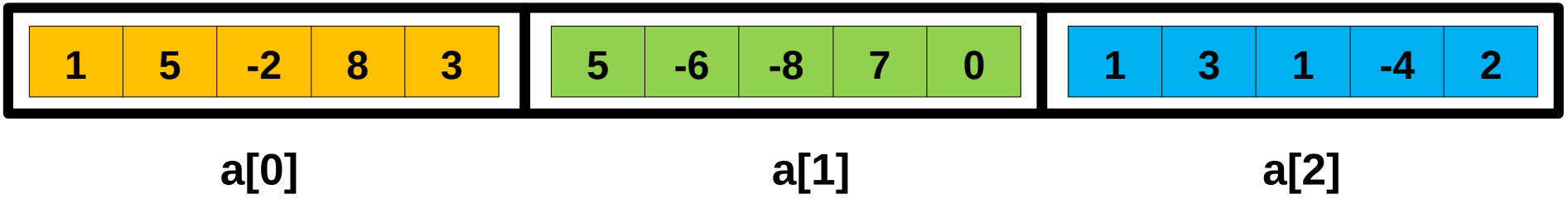
**Reprezentarea matricelor in memorie = tablouri de tablouri**

int a[3][5]

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 1 | 5 | -2 | 8 | 3 |
| 1 | 5 | -6 | -8 | 7 | 0 |
| 2 | 1 | 3 | 1 | -4 | 2 |

| 1 | 5 | -2 | 8 | 3 | 5 | -6 | -8 | 7 | 0 | 1 | 3 | 1 | -4 | 2 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

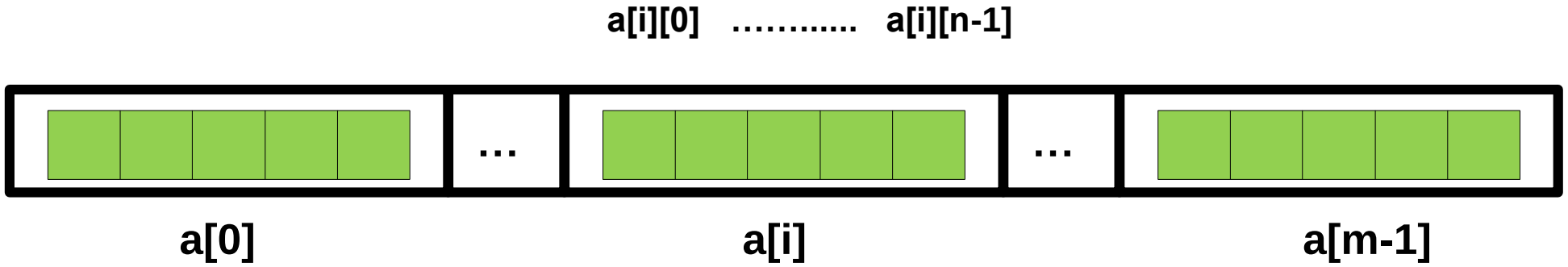
a[0][0] ……. a[0][4] a[1][0] …… a[2][4]



# 1. Tablouri bidimensionale – notiuni teoretice

**Reprezentarea matricelor in memorie = tablouri de tablouri**

Generalizare: a matrice cu m linii si n coloane




**Numele unui tablou este un pointer constant catre primul sau element**

int v[100]; v ***identic cu*** &v[0];
float a[4][6]; a ***identic cu*** &a[0][0];

# 1. Tablouri bidimensionale – notiuni teoretice

## Varianta C / C++

**Cantitatea de memorie necesara pentru inregistrarea unui tablou este direct proportionala cu tipul si marimea sa.**

tip nume [dimensiune1][dimensiune2] →
**sizeof(nume) = sizeof (tip) * dimensiune1 * dimensiune2;**

```
double tab [5][10] ;

int a[5][10];

char b1[5][10];

printf("Stocarea unui tablou de elemente double = %d octeti\n",sizeof(tab));
printf("Stocarea unui tablou de elemente int = %d octeti\n",sizeof(a));
printf("Stocarea unui tablou de elemente char = %d octeti\n",sizeof(b1));
```

```
C:\Users\Ank\Desktop\testSizeof\bin\Debug\testSizeof.exe
Stocarea unui tablou de elemente double = 400 octeti
Stocarea unui tablou de elemente int = 200 octeti
Stocarea unui tablou de elemente char = 50 octeti
```

# 1. Tablouri bidimensionale – notiuni teoretice

```
int a[10][10], n, m;
```

## Citirea unei matrice

```
cin>>n>>m;
for(i=0; i<n; i++)
   for(j=0; j<m; j++)
      cin>>a[i][j];
```

## Interschimbarea liniei x cu linia y

```
for(j=0; j<m; j++)
{
    aux = a[x][j];
    a[x][j] = a[y][j];
    a[y][j] = aux;
}
```

## Afisarea unei matrice

```
for(i=0; i<n; i++)
{
   for(j=0; j<m; j++)
      cout<<a[i][j]<<" ";
   cout<<endl;
}
```

## Stergerea liniei x dintr-o matrice

```
for(i=x; i<n-1; i++)
      for(j=0; j<m; j++)
         a[i][j] = a[i+1][j];
n--;
```

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.1 – Puncte "șa"

***Se citeste o matrice cu n linii si m coloane, cu elemente distincte. Sa se afiseze punctele "șa" din matrice. Un punct "șa" este acel element care este minim pe linia sa si maxim pe coloana.***

**Exemplu**

int a[3][5]

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 10 | 5 | -2 | 8 | 33 |
| 1 | 4 | -8 | -6 | 7 | 0 |
| 2 | 3 | 13 | 2 | 4 | 9 |

Punct "**șa**": a[2][2]

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.1 – Puncte "șa"

***Matrice cu elemente distincte***

***Sugestie de implementare:***

1.consideram un subprogram care sa returneze **pozitia pe care se afla elementul minim de pe o linie**;

2.consideram un subprogram care sa returneze **pozitia pe care se afla elementul maxim de pe o coloana;**

3.Parcurgem matricea linie cu linie; pe fiecare linie cautam pozitia minimului. Daca aceasta coincide cu pozitia maximului de pe coloana minimului, atunci afisam elementul sa.

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.1 – Puncte "șa"

***Matrice cu elemente distincte***

### Pozitia elementului minim pe o linie

```
int poz_min_linie(int lin, int m)
{
    int j,p=0;
    for(j=1; j<m; j++)
        if (a[lin][j]<a[lin][p])
            p = j;
    return p;
}
```

### Pozitia elementului maxim pe o coloana

```
int poz_max_coloana(int col, int n)
{
    int i,p=0;
    for(i=1; i<n; i++)
        if (a[i][col]>a[p][col])
            p = i;
    return p;
}
```

### Afisarea elementului "sa"

```
for(i=0; i<n; i++)
{
    j = poz_min_linie(i,m);
    if (i==poz_max_coloana(j,n))
        cout<<i<<" "<<j<<'\n';
}
```

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.1 – Puncte "șa"

***Cum ati rezolva problema in cazul unei matrice cu n linii si m coloane cu elemente nedistincte?***

**Exemplu**

int a[3][5]

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 1 | 4 | 1 | 5 | 2 |
| 1 | 3 | 5 | 3 | 7 | 3 |
| 2 | 2 | 3 | 0 | 1 | 2 |

Puncte "**șa**": a[1][0], a[1][2], a[1][4]

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.1 – Puncte "șa"

***Matrice cu elemente nedistincte***

***Sugestie de implementare:***

1.consideram un subprogram care returneaza **elementul minim de pe o linie**;
2.Consideram vectorul min_lin[] care sa contina valorile minime de pe fiecare linie a matricei
3.consideram un subprogram care returneaza **elementul maxim de pe o coloana;**
4.Consideram un vector max_col[] care sa contina valorile maxime de pe fiecare coloana a matricei;
5.Parcurgem matricea element cu element; daca a[i][j] coincide cu valoarea de pe pozitia i din vectorul min_lin[] si cu valoarea de pe pozitia j din vectorul max_col[], atunci afisam elementul sa.

| | | | | | min_lin: |
|---|---|---|---|---|---|
| 1 | 4 | 1 | 5 | 2 | 1 |
| 3 | 5 | 3 | 7 | 3 | 3 |
| 2 | 3 | 0 | 1 | 2 | 0 |
| max_col: 3 | 5 | 3 | 7 | 3 | |

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.1 – Puncte "șa"

***Matrice cu elemente nedistincte***

### Elementul minim pe o linie

```
int  min_linie(int lin, int m)
{
    int j,p=0;
    for(j=1; j<m; j++)
        if (a[lin][j]<a[lin][p])
            p = j;
    return a[lin][p];
}
```

### Elementul maxim pe o coloana

```
int  max_coloana(int col, int n)
{
    int i,p=0;
    for(i=1; i<n; i++)
        if (a[i][col]>a[p][col])
            p = i;
    return a[p][col];
}
```

### Afisarea elementului "sa"

```
for(i=0; i<n; i++)
    min_lin[i] = min_linie(i,m);

for(j=0; j<m; j++)
    max_col[j] =max_coloana(j,n);

for(i=0; i<n; i++)
    for(j=0; j<m; j++)
        if(a[i][j]==min_lin[i] && a[i][j]==max_col[j])
            cout<<i+1<<" "<<j+1<<endl;
```

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.2 – Ordonare diagonala

***Se citeste o matrice patratica de dimensiune n x n. Să se sorteze crescator valorile aflate pe diagonala principală, astfel incat, pe fiecare linie si pe fiecare coloana sa ramana aceleasi valori (liniile pot fi insa intr-o alta intr-o alta ordine in noua matrice, la fel si coloanele); altfel spus, sa se sorteze crescator valorile de pe diagonala principala prin interschimbari de linii si de coloane.***

**Exemplu** int a[4][4]

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 9 | 1 | 7 | 8 |
| 1 | 2 | 6 | 4 | 3 |
| 2 | 5 | 2 | 1 | 8 |
| 3 | 7 | 6 | 4 | 4 |

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 1 | 8 | 2 | 5 |
| 1 | 4 | 4 | 6 | 7 |
| 2 | 4 | 3 | 6 | 2 |
| 3 | 7 | 8 | 1 | 9 |

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.2 – Ordonare diagonala

***Se citeste o matrice patratica de dimensiune n x n. Să se sorteze crescator valorile aflate pe diagonala principală, astfel incat, pe fiecare linie si pe fiecare coloana sa ramana aceleasi valori (liniile pot fi insa intr-o alta intr-o alta ordine in noua matrice, la fel si coloanele); altfel spus, sa se sorteze crescator valorile de pe diagonala prinicipala prin interschimbari de linii si de coloane***

***Sugestie de implementare:***

1.consideram un subprogram care sa interschimbe doua linii;
2.consideram un subprogram care sa interschimbe doua coloane;
3.ordonam crescator elemntele de pe diagonala principala utilizand algoritmul de **selectie a minimului** (La fiecare iteratie i se interschimba linia i cu linia corespunzatoare minimului, respectiv coloana i cu coloana minimului).

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.2 – Ordonare diagonala

### Interschimbarea liniei x cu linia y

```
void swap_linie(int x, int y)
{
    int i, aux;
    for(i=0; i<n; i++)
    {
        aux = a[x][i];
        a[x][i] = a[y][i];
        a[y][i] = aux;
    }
}
```

### Interschimbarea coloanei x cu coloana y

```
void swap_coloana(int x, int y)
{
    int i, aux;
    for(i=0; i<n; i++)
    {
        aux = a[i][x];
        a[i][x] = a[i][y];
        a[i][y] = aux;
    }
}
```

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.2 – Ordonare diagonala

**Ordonarea diagonalei principale folosind algoritmul de sortare prin selectia minimului**

```
for(i=0; i<n-1; i++)
{
    minim = i;
    for(j = i+1; j<n; j++)
        if (a[j][j] < a[minim][minim])
            minim = j;
    swap_linie(i,minim);
    swap_coloana(i,minim);
}
```

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.3 – Diagonalele si paralelele acestora

***Se citeste o matrice patratica de dimensiune n x n. Să se afiseze elementele de pe diagonalele matricei si de pe liniile paralele acestora.***

**Exemplu**

int a[4][4]

| | 0 | 1 | 2 | 3 |
|---|---|---|---|---|
| 0 | 9 | 1 | 7 | 8 |
| 1 | 2 | 6 | 4 | 3 |
| 2 | 5 | 2 | 1 | 8 |
| 3 | 7 | 6 | 4 | 4 |

**Diagonale principale**

| | | | |
|---|---|---|---|
| 8 | | | |
| 7 | 3 | | |
| 1 | 4 | 8 | |
| 9 | 6 | 1 | 4 |
| 2 | 2 | 4 | |
| 5 | 6 | | |
| 7 | | | |

**Diagonale secundare**

| | | | |
|---|---|---|---|
| 9 | | | |
| 1 | 2 | | |
| 7 | 6 | 5 | |
| 8 | 4 | 2 | 7 |
| 3 | 1 | 6 | |
| 8 | 4 | | |
| 4 | | | |

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.3 – Diagonalele si paralelele acestora

***Sugestie de implementare:***

1.Se considera k numarul paralelei cu diagonala principala, respectiv secundara; prima paralela la diagonala principala considera ultimul element de pe prima linie (dreapta sus), iar prima paralela la diagonala secundara considera primul element de pe prima linie (stanga sus);
2.Varianta **$O(n^3)$**: pentru fiecare k se parcurge matricea element cu element si se verifica conditia de paralela la diagonala principala;
3.Varianta **$O(n^3)$**: pentru fiecare k se parcurge matricea element cu element si se verifica conditia de paralela la diagonala secundara;
4.Varianta **$O(n^2)$**: pentru fiecare k se considera elementele de pe linia i si paralela k la diagonala principala (**indicele de coloana este inlocuit cu formula corespunzatoare**);
5.Varianta **$O(n^2)$**: pentru fiecare k se considera elementele de pe linia i si paralela k la diagonala secundara (**indicele de coloana este inlocuit cu formula corespunzatoare**).

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.3 – Diagonalele si paralelele acestora

$O(n^3)$

### Paralele la diagonala principala

```
//k = a cata diagonala parcurgem
for (int k = 0; k<2*n-1; k++)
{
   for(i=0; i<n; i++)
      for(j=0; j<n; j++)
         if (j-i == n - k - 1) cout<<a[i][j]<<" ";
   cout<<'\n';
}
```

### Paralele la diagonala secundara

```
//k = a cata diagonala parcurgem
for (int k = 0; k<2*n-1; k++)
{
   for(i=0; i<n; i++)
      for(j=0; j<n; j++)
         if (i+j == k) cout<<a[i][j]<<" ";
   cout<<'\n';
}
```

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.3 – Diagonalele si paralelele acestora

$O(n^2)$

### Paralele la diagonala principala

```
//k = a cata diagonala parcurgem
for (int k = 0; k<n; k++)
{
    for(i=0; i<k+1; i++)
            cout<<a[i][n - k - 1 + i]<<" ";
    cout<<'\n';
}

for (int k = n; k<2*n-1; k++)
{
    for(i=k-n+1; i<n; i++)
            cout<<a[i][n - k - 1 + i]<<" ";
    cout<<'\n';
}
```

### Paralele la diagonala secundara

```
//k = a cata diagonala parcurgem
for (int k = 0; k<n; k++)
{
    for(i=0; i<k+1; i++)
        cout<<a[i][k-i]<<" ";
    cout<<'\n';
}

for (int k = n; k<2*n-1; k++)
{
    for(i=k-n+1; i<n; i++)
        cout<<a[i][k-i]<<" ";
    cout<<'\n';
}
```

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.4 – Atac regine

### Cerința

Pe o tablă de șah de dimensiune $n$ se află $m$ regine. O regină atacă o altă regină dacă cele două se află pe aceeași linie, coloană sau diagonală și între ele nu se află alte regine. Determinați numărul maxim $p$ de regine care sunt atacate de o aceeași regină și numărul $q$ de regine care atacă $p$ alte regine.

### Date de intrare

Fișierul de intrare `regine.in` conține pe prima linie numerele $n$ $m$; următoarele $m$ linii conțin perechi $i$ $j$ reprezentând linia și coloana unde se află poziționată o regină.

https://www.pbinfo.ro/?pagina=probleme&id=602

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.4 – Atac regine

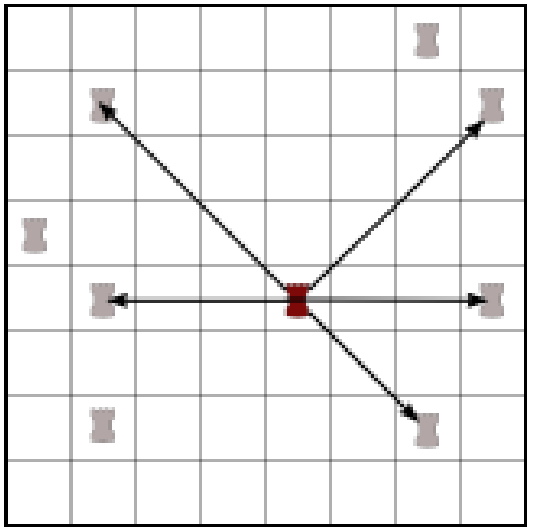
*regine.in*

```
8 9
1 7
2 2
2 8
4 1
5 2
5 5
5 8
7 2
7 7
```

*regine.out*

```
5 1
```

Explicație



https://www.pbinfo.ro/?pagina=probleme&id=602

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.4 – Atac regine

### *Caz 1. numarul de regine este mult mai mare decat n – Complexitate $O(n^2)$*

***Sugestie de implementare:***

1. construim matricea a cu semnificatia
   a[x][y] = 0, daca pe pozitia (x,y) nu este plasata o regina
   a[x][y] = numarul reginei aflata pe linia x si coloana y, altfel

*regine.in*

```
8 9
1 7
2 2
2 8
4 1
5 2
5 5
5 8
7 2
7 7
```



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | 1 | |
| | 2 | | | | | | 3 |
| | | | | | | | |
| 4 | | | | | | | |
| | 5 | | | 6 | | | 7 |
| | | | | | | | |
| | 8 | | | | | 9 | |
| | | | | | | | |

matricea a
celulele libere memoreaza 0

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.4 – Atac regine

### *Caz 1. numarul de regine este mult mai mare decat n – Complexitate $O(n^2)$*

***Sugestie de implementare:***

1. construim matricea a cu semnificatia
   a[x][y] = 0, daca pe pozitia (x,y) nu este plasata o regina
   a[x][y] = numarul reginei aflata pe linia x si coloana y;
2. construim vectorul atac cu semnificatia
   atac[i]=cate regine sunt atacate de regina i (initial 0);
3. pentru fiecare regina stabilim ce regine **de pe aceeasi line** ataca, **parcurgand matricea linie cu linie;**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | 1 | |
| | 2 | | | | | | 3 |
| | | | | | | | |
| 4 | | | | | | | |
| | 5 | | | 6 | | | 7 |
| | | | | | | | |
| | 8 | | | | | 9 | |
| | | | | | | | |

**De exemplu, când parcurgem linia 5, găsim următoarele perechi de numere pozitive consecutive (care au doar 0 intre ele):**
**(5,6) ⇒ crește atac[5] si atac[6]**
**(6,7) ⇒ crește atac[6] si atac[7]**

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.4 – Atac regine

### *Caz 1. numarul de regine este mult mai mare decat n – Complexitate $O(n^2)$*

***Sugestie de implementare:***

1. construim matricea a cu semnificatia
   a[x][y] = 0, daca pe pozitia (x,y) nu este plasata o regina
   a[x][y] = numarul reginei aflata pe linia x si coloana y;
2. construim vectorul atac cu semnificatia
   atac[i]=cate regine sunt atacate de regina i (initial 0);
3. pentru fiecare regina stabilim ce regine **de pe aceeasi line** ataca, **parcurgand matricea linie cu linie;**
4. pentru fiecare regina stabilim ce regine **de pe aceeasi coloana** ataca, **parcurgand matricea coloana cu coloana;**
5. pentru fiecare regina stabilim ce regine **de pe aceeasi diagonala paralela cu cea principala** ataca, parcurgand matricea paralel cu diagonala principala;
6. pentru fiecare regina stabilim ce regine de pe aceeasi **diagonala paralela cu cea secundara** ataca, parcurgand matricea paralel cu diagonala secundara
7. Parcurgem vectorul atac[] pentru gasirea elementului maxim si a frecventei acestuia.

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.4 – Atac regine

### *Caz 1. numarul de regine este mult mai mare decat n – Complexitate O($n^2$)*

***Actualizarea vectorului "atac" prin considerarea conflictelor de pe aceeasi linie***

| Regina | atac |
|---|---|
| 1 | 0 |
| 2 | 1 |
| 3 | 1 |
| 4 | 0 |
| 5 | 1 |
| 6 | 2 |
| 7 | 1 |
| 8 | 1 |
| 9 | 1 |

```
void numara_conflicte_linie (int atac[501], int a[101][101], int n)
{
    int i,j,poz;
    for(i=1; i<=n; i++)
    {
        poz=0;// coloana ultimei regine de pe linia i
        for(j=1; j<=n; j++)
            if (a[i][j]!=0) //este regina
                if(poz==0)
                    poz=j; //prima regina de pe linie
                else
                {
                    //regina anterioara de pe linia i este pe coloana poz
                    atac[a[i][poz]]++;
                    atac[a[i][j]]++;
                    poz=j;
                }
    }
}
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | 1 | |
| | 2 | | | | | | 3 |
| | | | | | | | |
| 4 | | | | | | | |
| | 5 | | | 6 | | | 7 |
| | | | | | | | |
| | 8 | | | | | 9 | |
| | | | | | | | |

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.4 – Atac regine

### *Caz 1. numarul de regine este mult mai mare decat n – Complexitate $O(n^2)$*

***Actualizarea vectorului "atac" prin considerarea conflictelor de pe aceeasi coloana***

| Regina | Atac |
|---|---|
| 1 | 1 |
| 2 | 2 |
| 3 | 2 |
| 4 | 0 |
| 5 | 3 |
| 6 | 2 |
| 7 | 2 |
| 8 | 2 |
| 9 | 2 |

```
void numara_conflicte_coloana(int atac[501],int
{                                  a[101][101],int n)
    int i,j,poz;
    for(j=1; j<=n; j++)
    {
        poz=0;
        for(i=1; i<=n; i++)
            if (a[i][j]!=0)
                if(poz==0)
                    poz=i;
                else
                {
                    atac[a[poz][j]]++;
                    atac[a[i][j]]++;
                    poz=i;
                }
    }
}
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | 1 | |
| | 2 | | | | | | 3 |
| | | | | | | | |
| 4 | | | | | | | |
| | 5 | | | 6 | | | 7 |
| | | | | | | | |
| | 8 | | | | | 9 | |
| | | | | | | | |

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.4 – Atac regine

### *Caz 1. numarul de regine este mult mai mare decat n – Complexitate $O(n^2)$*

***Actualizarea vectorului "atac" prin considerarea conflictelor de pe diagonalele paralele cu diagonala principala***

| Regina | Atac |
|---|---|
| 1 | 2 |
| 2 | 3 |
| 3 | 3 |
| 4 | 1 |
| 5 | 4 |
| 6 | 4 |
| 7 | 2 |
| 8 | 2 |
| 9 | 3 |

```
void numara_conflicte_diag_principala(int atac[501], int a[101][101],
                                      int n)
{
   int i,j,poz,k;

   for (k = 0; k<n; k++)
   {
      poz=0;
      for(i=1; i<=k+1; i++)
      {
         int j = n - k - 1 + i;
         if (a[i][j]!=0)
            if(poz==0)
               poz=i;
            else
            {
               atac[a[poz][n - k - 1 + poz]]++;
               atac[a[i][j]]++;
               poz=i;
            }

      }
   }
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | 1 | |
| | 2 | | | | | | 3 |
| | | | | | | | |
| 4 | | | | | | | |
| | 5 | | | 6 | | | 7 |
| | | | | | | | |
| | 8 | | | | | 9 | |
| | | | | | | | |

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.4 – Atac regine

### *Caz 1. numarul de regine este mult mai mare decat n – Complexitate $O(n^2)$*

***Actualizarea vectorului "atac" prin considerarea conflictelor de pe diagonalele paralele cu diagonala principala***

| Regina | Atac |
|---|---|
| 1 | 2 |
| 2 | 3 |
| 3 | 3 |
| 4 | 1 |
| 5 | 4 |
| 6 | 4 |
| 7 | 2 |
| 8 | 2 |
| 9 | 3 |

```
//se poate translata de la 0 scazand n
   for (k = n; k<2*n-1; k++)
   {
      poz=0;
      for(i=k-n+2; i<=n; i++)
      {
         int j=n - k - 1 + i;
         if (a[i][j]!=0)
            if(poz==0)
               poz=i;
            else
            {
               atac[a[poz][n - k - 1 + poz]]++;
               atac[a[i][j]]++;
               poz=i;
            }
      }
   }

}
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | 1 | |
| | 2 | | | | | | 3 |
| | | | | | | | |
| 4 | | | | | | | |
| | 5 | | | 6 | | | 7 |
| | | | | | | | |
| | 8 | | | | | 9 | |
| | | | | | | | |

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.4 – Atac regine

### *Caz 1. numarul de regine este mult mai mare decat n – Complexitate O($n^2$)*

***Actualizarea vectorului "atac" prin considerarea conflictelor de pe diagonalele paralele cu diagonala secundara***

| Regina | Atac |
|---|---|
| 1 | 2 |
| 2 | 3 |
| 3 | 4 |
| 4 | 1 |
| 5 | 4 |
| 6 | 5 |
| 7 | 2 |
| 8 | 2 |
| 9 | 3 |

```
void numara_conflicte_diag_secundara(int atac[501], int a[101][101],
                                     int n)
{
    int i,j,poz,k;
    for (int k = 0; k<n; k++)
    {
        poz=0;
        for(i=1; i<=k+1; i++)
        {
            int j=k+2-i;
            if (a[i][j]!=0)
                if(poz==0)
                    poz=i;
                else
                {
                    atac[a[poz][k+2-poz]]++;
                    atac[a[i][j]]++;
                    poz=i;
                }
        }

    }
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | 1 | |
| | 2 | | | | | | 3 |
| | | | | | | | |
| 4 | | | | | | | |
| | 5 | | | 6 | | | 7 |
| | | | | | | | |
| | 8 | | | | | 9 | |
| | | | | | | | |

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.4 – Atac regine

### *Caz 1. numarul de regine este mult mai mare decat n – Complexitate $O(n^2)$*

***Actualizarea vectorului "atac" prin considerarea conflictelor de pe diagonalele paralele cu diagonala secundara***

| Regina | Atac |
|---|---|
| 1 | 2 |
| 2 | 3 |
| 3 | 4 |
| 4 | 1 |
| 5 | 4 |
| 6 | 5 |
| 7 | 2 |
| 8 | 2 |
| 9 | 3 |

```
for (int k = n; k<2*n-1; k++)
{
    poz=0;
    for(i=k-n+2; i<=n; i++)
    {
        int j=k+2-i;
        if (a[i][j]!=0)
            if(poz==0)
                poz=i;
            else
            {
                atac[a[poz][k+2-poz]]++;
                atac[a[i][j]]++;
                poz=i;
            }
    }
}
```

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | 1 | |
| | 2 | | | | | | 3 |
| | | | | | | | |
| 4 | | | | | | | |
| | 5 | | | 6 | | | 7 |
| | | | | | | | |
| | 8 | | | | | 9 | |
| | | | | | | | |

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.4 – Atac regine

### *Caz 1. numarul de regine este mult mai mare decat n – Complexitate $O(n^2)$*

***Constructia matricei initiale***

```
ifstream f("regine.in");
f>>n>>m;
for(i = 1; i <= m; i++)
{
    f>>x>>y;
    a[x][y]=i;
}
f.close();
```

***Actualizarea vectorului "atac"***

```
numara_conflicte_linie(atac,a,n);
numara_conflicte_coloana(atac,a,n);
numara_conflicte_diag_principala(atac,a,n);
numara_conflicte_diag_secundara(atac,a,n);
```

***Gasirea reginei cu cele mai multe atacuri***

```
//p=maximul din vectorul atac
//q=frecventa maximului
p = q = 1;
for(i = 1; i <=m; i++)
    if(atac[i] > p)
    {
        p = atac[i];
        q = 1;
    }
    else if(atac[i] == p) q++;

cout<<p<<" "<<q;
```

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.4 – Atac regine

### *Caz 2. numarul de regine este mult mai mic decat n – Complexitate $O(m^2)$*

***Sugestie de implementare:***

1. consideram un vector de structuri “r”, unde fiecare componenta retine coordonatele unei regine pe tabla de sah;
2. sortam vectorul in ordinea crescatoare a absciselor pozitiilor reginelor; **la abscise egale, sortam dupa ordonata**;
3. construim vectorul atac cu semnificatia
   atac[i]=cate regine sunt atacate de regina i;
4. pentru fiecare regina “i” cautam **prima regina aflata in dreapta sa**, pe aceeasi linie = **prima regina care ii urmeaza in vectorul r** care se afla pe aceeasi linie ;
5. pentru fiecare regina “i” cautam **prima regina aflata sub ea**, pe aceeasi coloana; = **prima regina care ii urmeaza in vectorul r** care se afla pe aceeasi coloana ;
6. pentru fiecare regina “i” cautam **prima regina aflata pe aceeasi diag. principala;**
7. pentru fiecare regina “i” cautam **prima regina aflata pe aceeasi diag. secundara**;
8. Parcurgem vectorul atac[] pentru gasirea elementului maxim si a frecventei acestuia.

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.4 – Atac regine

### *Caz 2. numarul de regine este mult mai mic decat n – Complexitate O($m^2$)*

***Exemplificare modificare vector atac dupa iteratia pentru Regina 1***

| R[i].x | R[i].y | Atac_lin | Atac_col | Atac_dp | Atac_ds | Total anterior | Total Actual |
|---|---|---|---|---|---|---|---|
| 1 | 7 | | 1 | 1 | | 0 | 2 |
| 2 | 2 | | | | | 0 | 0 |
| 2 | 8 | | | | | 0 | 1 |
| 4 | 1 | | | | | 0 | 0 |
| 5 | 2 | | | | | 0 | 0 |
| 5 | 5 | | | | | 0 | 0 |
| 5 | 8 | | | | | 0 | 0 |
| 7 | 2 | | | | | 0 | 0 |
| 7 | 7 | | | | | 0 | 1 |

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.4 – Atac regine

### *Caz 2. numarul de regine este mult mai mic decat n – Complexitate O($m^2$)*

***Exemplificare modificare vector atac dupa iteratia pentru Regina 2***

| R[i].x | R[i].y | Atac_lin | Atac_col | Atac_dp | Atac_ds | Total anterior | Total Actual |
|---|---|---|---|---|---|---|---|
| 1 | 7 | | | | | 2 | 2 |
| 2 | 2 | 1 | 1 | 1 | | 0 | 3 |
| 2 | 8 | | | | | 1 | 2 |
| 4 | 1 | | | | | 0 | 0 |
| 5 | 2 | | | | | 0 | 1 |
| 5 | 5 | | | | | 0 | 1 |
| 5 | 8 | | | | | 0 | 0 |
| 7 | 2 | | | | | 0 | 0 |
| 7 | 7 | | | | | 1 | 1 |

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.4 – Atac regine

### *Caz 2. numarul de regine este mult mai mic decat n – Complexitate O($m^2$)*

***Actualizarea vectorului "atac" prin considerarea primului conflict aparut pe aceeasi linie***

```
for(i = 0; i < m; i++)
{

    pl = pc = pdp = pds = 0;
    /*variabile flag pentru a marca momentul in care intalnim o regina
     atacata de regina i: pe linie la dreapta, pe coloana in jos,
     pe diagonala principala, respectiv secundara in jos*/

    for(j = i + 1; j < m; j++)
    {
        //prima regina atacata de regina i pe linie
        if((r[i].x == r[j].x) && (pl == 0))
        {

            atac[i]++;
            atac[j]++;
            pl = 1;
        }
```

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.4 – Atac regine

### *Caz 2. numarul de regine este mult mai mic decat n – Complexitate $O(m^2)$*

***Actualizarea vectorului "atac" prin considerarea primului conflict aparut pe aceeasi coloana, respectiv pe aceeasi diagonala principala si secundara***

```
    else
        if((r[i].y == r[j].y) && (pc == 0))
        {

            atac[i]++;
            atac[j]++;
            pc = 1;
        }
        else
            if((r[i].x - r[i].y == r[j].x - r[j].y) && (pdp == 0))
            {

                atac[i]++;
                atac[j]++;
                pdp = 1;
            }
            else
                if((r[i].x + r[i].y == r[j].x + r[j].y) && (pds == 0))
                {

                    atac[i]++;
                    atac[j]++;
                    pds = 1;
                }

    if(pl + pc + pdp + pds == 4) break;
}
```

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.4 – Atac regine

### *Caz 2. numarul de regine este mult mai mic decat n – Complexitate O($m^2$)*

***Constructia vectorului de pozitii***

```
f>>n>>m;
for(i = 0; i < m; i++){
    f>>r[i].x>>r[i].y;
}
f.close();
```

***Sortarea vectorului de pozitii***

```
int cmpRegine(Regina  r1 , Regina r2)
{
    if(r1.x<r2.x) return 1;
    if(r1.x == r2.x)
        if(r1.y < r2.y) return 1;
    return 0;
}
```

```
sort(r, r+m,  cmpRegine);
```

***Gasirea reginei cu cele mai multe atacuri***

```
//p=maximul din vectorul atac
//q=frecventa maximului
p = q = 0;
for(i = 0; i < m; i++)
    if(atac[i] > p)
    {
        p = atac[i];
        q = 1;
    }
    else
        if(atac[i] == p) q++;

cout<<p<<" "<<q;
```

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.5 – Exista valoare in matrice

***Se consideră o matrice în care fiecare linie și fiecare coloană este sortată strict crescător.***
***Să se determine dacă o valoare dată se găsește în matrice sau nu.***

**Exemplu**

int a[3][5]

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| 4 | 6 | 8 | 10 | 12 |
| 5 | 7 | 12 | 24 | 29 |

Valoarea 10 se afla in matrice.
Valoarea 9 nu se afla in matrice

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.5 – Exista valoare in matrice

**O(nm)**

- ***rezolvare1*** – **Cautarea standard a unui element intr-o matrice**
  - se parcurge matricea element cu element si verifica existenta valorii cautate in matrice;

```
void rezolvare1(int &gasit, int x)
{
    int i,j;
    gasit = 0;
    for(i=0; i<n; i++)
        for(j=0; j<m; j++)
            if (a[i][j] == x)
                gasit = 1;
}
```

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.5 – Exista valoare in matrice

**O(n log m)**

***rezolvare2 - Cautarea unui element intr-o matrice folosind, pe fiecare linie, cautarea binara, intrucat fiecare linie e ordonata crescator.***

```
int caut_binar( int v[], int s, int d, int x)
{
    int m;
    while (s<=d)
    {
        m = (s+d)/2;
        if (x == v[m]) return m;
        if (x<v[m])
            d = m - 1;
        else
            s = m + 1;
    }
    return -1;
}
```

```
void rezolvare2(int &gasit, int x)
{
    gasit = 0;
    for(i=0; i<n; i++)
    {
        if (caut_binar(a[i],0,m-1,x)!= -1) gasit = 1;
    }
}
```

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.5 – Exista valoare in matrice

**O(n log m)**

***rezolvare3 - Cautarea unui element intr-o matrice folosind, pe fiecare linie, cautarea binara, intrucat fiecare linie e ordonata crescator. Optimizare: ignorarea acelor linii care sigur nu contin elementul x (cele care incep cu elemente > x si cele care se termina cu elemente < x).***

| 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| 4 | 6 | 8 | 10 | 12 |
| 5 | 7 | 12 | 24 | 29 |
| 11 | 13 | 15 | 28 | 30 |
| 40 | 41 | 43 | 45 | 51 |

De exemplu, **valoarea 10 se poate afla doar pe liniile 2 si 3**, nu are sens sa o cautam pe:
-linia 1 – deoarece are pe ultima coloana valoarea 5,
deci cea mai mare valoare de pe linia 1 este 5,
-linia 4 – deoarece are pe prima linie valoarea 11,
deci cea mai mica valoare de pe linie este 11,
-linia 5 – cea mai mica valoare de pe linie este 40

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.5 – Exista valoare in matrice

**O(n log m)**

***rezolvare3 - Cautarea unui element intr-o matrice folosind, pe fiecare linie, cautarea binara, intrucat fiecare linie e ordonata crescator. Optimizare: ignorarea acelor linii care sigur nu contin elementul x (cele care incep cu elemente > x si cele care se termina cu elemente < x).***

```
void restrictionare_linii(int &j, int &k,int x)
{
// daca ultimele elemente < x sau primele su
    for (k = n-1; k>=0; k--)
        if (a[k][0] <= x) break;

    for (j = 0; j<=k; j++)
        if (a[j][m-1] >= x) break;
}
```

```
void rezolvare3(int &gasit, int x)
{
    int i,j,k;
    restrictionare_linii(j,k,x);
    gasit = 0;
    for(i=j; i<=k; i++)
    {
        if (caut_binar(a[i],0,m-1,x)!= -1) gasit = 1;
    }
}
```

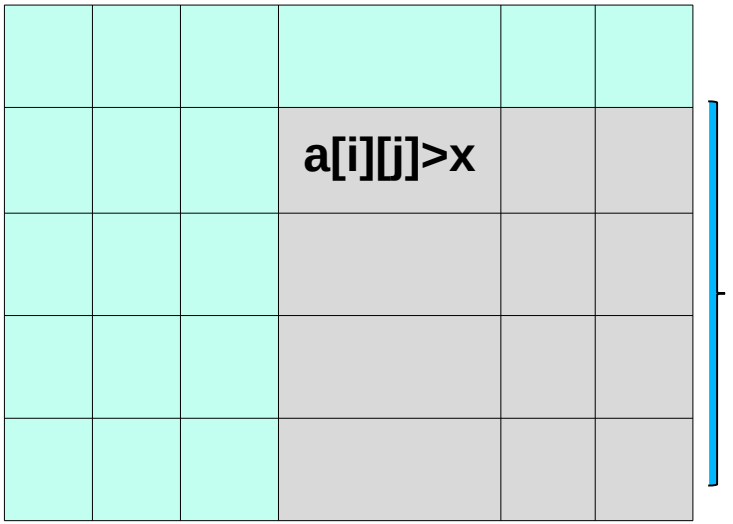
# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.5 – Exista valoare in matrice

**O(n + m)**

***rezolvare4 – restrictionam mai mult decat in rezolvarea3 zona in care cautam valoarea x, pe baza urmatoarelor observatii:***

1. Daca un element de pe pozitia (i,j) este mai mare (strict) decat x, atunci toate elementele aflate pe pozitii (r,s) cu r ≥ i si s ≥ j sunt mai mari decat x:




Toate elementele din aceasta zona vor fi mai mari decat x ⇒
**x nu mai trebuie cautat in aceasta zona**

# 2. Tablouri bidimensionale – Aplicatii

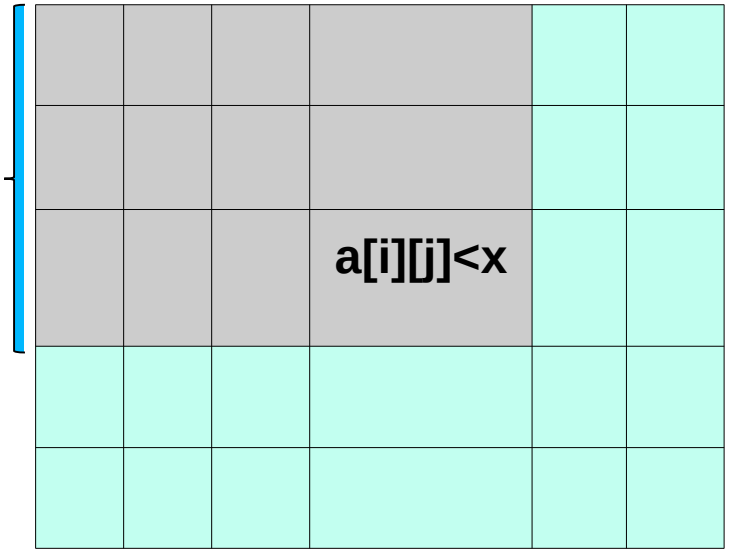
## Aplicatie 2.5 – Exista valoare in matrice

**O(n + m)**

***rezolvare4 – restrictionam mai mult decat in rezolvarea3 zona in care cautam valoarea x, pe baza urmatoarelor observatii:***

2. Daca un element de pe pozitia (i,j) este mai mic (strict) decat x, atunci toate elementele aflate pe pozitii (r,s) cu r ≤ i si s ≤ j sunt mai mici decat x:

Toate elementele din aceasta zona vor fi mai **mici** decat x ⇒
**x nu mai trebuie cautat in aceasta zona**



# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.5 – Exista valoare in matrice

**O(n + m)**

***rezolvare4 – restrictionam mai mult decat in rezolvarea3 zona in care cautam valoarea x, pe baza urmatoarelor observatii:***

3. Din observatiile anterioare rezulta ca putem parcurge matricea:
   - pe o linie de la dreapta la stanga pana intalnim un element a[i][j]≤ x ;
     daca elementul este chiar x, atunci ne oprim;
     altfel, din primele observatii, nu mai trebuie sa cautam pe coloanele > j (nici pe liniile ≤ i); astfel, putem cobori pe coloana j pe care am ajuns

| | | | | |
|---|---|---|---|---|
| 7 | 8 | 10 | 23 | 24 |
| 12 | 16 | 18 | 25 | 26 |
| 13 | 17 | 19 | 27 | 29 |
| 20 | 21 | 22 | 30 | 31 |
| 28 | 37 | 40 | 41 | 45 |

**Cautam x=21**

**presupunem liniile si coloanele numeroate de la 1**

-Parcurgem prima linie de la dreapta la stanga pana la primul element < 21, adica pana la a[1][3]=10

-In regiunea gri nu mai trebuie sa cautam, sigur sunt elemente > 21

⇒ **Coboram pe coloana 3**

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.5 – Exista valoare in matrice

**O(n + m)**

***rezolvare4 – restrictionam mai mult decat in rezolvarea3 zona in care cautam valoarea x, pe baza urmatoarelor observatii:***

3. Din observatiile anterioare rezulta ca putem parcurge matricea:
   - pe o linie de la dreapta la stanga pana intalnim un element a[i][j]≤ x ;
     daca elementul este chiar x, atunci ne oprim;
     altfel, din primele observatii, nu mai trebuie sa cautam pe coloanele > j (nici pe liniile ≤ i); astfel, putem cobori pe coloana j pe care am ajuns

   - **coboram pe coloana curenta j pana cand intalnim un element a[i][j] ≥ x ;**
     **daca elementul este chiar x, atunci ne oprim;**
     **altfel, din primele observatii, nu mai trebuie sa cautam pe linii < i (nici pe coloanele ≥ j);**

| | | | | |
|---|---|---|---|---|
| 7 | 8 | 10 | 23 | 24 |
| 12 | 16 | 18 | 25 | 26 |
| 13 | 17 | 19 | 27 | 29 |
| 20 | 21 | 22 | 30 | 31 |
| 28 | 37 | 40 | 41 | 45 |

**Cautam x=21**

**Coboram pe coloana 3** pana la a[4][3]=22>21

- In regiunea mov nu mai trebuie sa cautam, sigur sunt elemente < 21 (nici in cea gri, conform pasului anterior)

⇒ mergem la stanga pe linia 3

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.5 – Exista valoare in matrice

**O(n + m)**

***rezolvare4 – restrictionam mai mult decat in rezolvarea3 zona in care cautam valoarea x, pe baza urmatoarelor observatii:***

3. Din observatiile anterioare rezulta ca putem parcurge matricea:
   - pe o linie de la dreapta la stanga pana intalnim un element a[i][j]≤ x ;
     daca elementul este chiar x, atunci ne oprim;
     altfel, din primele observatii, nu mai trebuie sa cautam pe coloanele > j (nici pe liniile ≤ i); astfel, putem cobori pe coloana j pe care am ajuns

   - coboram pe coloana curenta j pana cand intalnim un element a[i][j] ≥ x ;
     daca elementul este chiar x, atunci ne oprim;
     altfel, din primele observatii, nu mai trebuie sa cautam pe linii < i (nici pe coloanele ≥ j); **este suficient sa mergem, din nou, pe linia curenta la stanga si reluam etapele**

| | | | | |
|---|---|---|---|---|
| **7** | **8** | **10** | **23** | **24** |
| **12** | **16** | **18** | **25** | **26** |
| **13** | **17** | **19** | **27** | **29** |
| **20** | **21** | **22** | **30** | **31** |
| **28** | **37** | **40** | **41** | **45** |

**Am gasit 21 ! ⇒ STOP**

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.5 – Exista valoare in matrice

**O(n + m)**

***rezolvare4 – restrictionam mai mult decat in rezolvarea3 zona in care cautam valoarea x, pe baza urmatoarelor observatii:***

Astfel, trebuie doar sa parcurgem matricea dupa regulile descrise **alternativ** pe linii de la dreapta la stanga si in jos pe coloane pana gasim x sau iesim din matrice sau ajungem la o linie/coloana restrictionata cu criteriul de la rezolvarea3

Spre exemplu, dacă valoarea căutată ar fi fost 15, traseul este urmatorul, si se ajunge la o linie restrictionata (care incepe cu un element>15)

| | | | | |
|---|---|---|---|---|
| 7 | 8 | 10 | 23 | 24 |
| 12 | 16 | 18 | 25 | 26 |
| 13 | 17 | 19 | 27 | 29 |
| 20 | 21 | 22 | 30 | 31 |
| 28 | 37 | 40 | 41 | 45 |

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.5 – Exista valoare in matrice

**O(n + m)**

***rezolvare4 –*** ***Parcurgerea alternativa a liniilor si coloanelor pana cand se gaseste numarul sau se atinge limita unei linii sau a unei coloane.***

```
void rezolvare4(int &gasit, int x)
{
    int j,k,linie,coloana;

    restrictionare_linii(j,k,x);
// toate liniile incep cu elem <= x si se termina cu elem >=x

    gasit = 0;
    linie = j;
    coloana = m-1;
    while (gasit == 0)
    {
        while (coloana >=0 && a[linie][coloana]>x) coloana--;
        if(coloana < 0) break;
        if (a[linie][coloana] == x){gasit = 1;break;}
        while (linie <= k && a[linie][coloana]<x) linie++;
        if (linie > k) break;
        if (a[linie][coloana] == x){gasit = 1;break;}
    }
}
```

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.6 – Element comun tuturor liniilor (daca exista)

***Se consideră o matrice în care fiecare linie este sortată crescător. Să se determine un element comun tuturor liniilor (dacă există).***

**Exemplu**

int a[4][5]

| | | | | |
|---|---|---|---|---|
| 1 | 7 | 12 | 15 | 20 |
| 3 | 4 | 7 | 11 | 16 |
| 2 | 3 | 4 | 7 | 15 |
| 7 | 8 | 9 | 10 | 11 |

Valoarea 7 este comuna.

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.6 – Element comun tuturor liniilor (daca exista)

***Sugestie de implementare:***

1. ***rezolvare()*** – foloseste algoritmul de cautare secventiala pentru gasirea unei valori pe o linie; aplica algoritmul pentru a cauta fiecare element de pe prima linie in fiecare linie din matrice;
2. ***rezolvare_binar()*** – optimizeaza subprogramul rezolvare() cautand binar o valoare intr-un vector;
3.– optimizeaza primele doua subprograme restrictionand limitele intervalelor de cautare (determinam pentru fiecare linie intre ce coloane este suficient sa cautam);
4. ***rezolvare3() – O(mn)*** *:*

    4.1 – se considera un vector col_crt[i] = retine ultima coloana de pe linia i pana la care trebuie cautat elementul comun;

    4.2 - determinam valoarea minima din col_crt - este o limita superioara pentru elementul comun;

    4.3 – scadem pt fiecare linie i col_crt[i] pana ajunge pe un element <=a[lin_min][col_crt[lin_min]] (mai la dreapta nu are sens sa cautam pe linie);

    4.4 – definim cnt = cate limite de coloane sunt egale cu valoarea minima a[lin_min][col_crt[lin_min]];

    4.5 – daca pentru fiecare linie elementul de pe coloana col_crt[i] este =a[lin_min][col_crt[lin_min]] atunci liniile au un element comun.

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.6 – Element comun tuturor liniilor (daca exista) $O(nm^2)$

***Cautarea secventiala***

```
int caut(int v[],int val){

    for(int j=0;j<m;j++)
        if(v[j]==val) return 1;

    return 0;
}
```

***Algoritmul complet de rezolvare***

```
void rezolvare()
{
    int comun,val,i;
    //verificam daca un element de pe prima linie se gaseste si pe celelalte
    for (int poz = 0; poz < m; poz++){
        comun = 1;
        val = a[0][poz];
        for (i = 1; i<n; i++)
            if (caut(a[i],val)==0){ comun = 0; break;}
        if (comun == 1) {cout<<"valoare comuna = "<<val<<endl; return;}
    }
    cout<<"nu exista valoare comuna"<<endl; return;
}
```

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.6 – Element comun tuturor liniilor (daca exista)

**O(nmlog m)**

***Cautarea binara***

***Algoritmul complet de rezolvare***

```
int caut_binar( int v[], int s, int d, int x)
{
    int m;
    while (s<=d)
    {
        m = (s+d)/2;
        if (x == v[m]) return m;
        if (x<v[m])
            d = m - 1;
        else
            s = m + 1;
    }
    return -1;
}
```

```
void rezolvare_binar()
{
    int comun,val,i;
    //verificam daca un element de pe prima linie se gaseste si pe celelalte
    for (int poz = 0; poz < m; poz++){
        comun = 1;
        val = a[0][poz];
        for (i = 1; i<n; i++)
            if (caut_binar(a[i],0,m-1,val)==-1){ comun = 0; break;}
        if (comun == 1) {cout<<"valoare comuna = "<<val<<endl; return;}
    }
    cout<<"nu exista valoare comuna"<<endl; return;
}
```

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.6 – Element comun tuturor liniilor (daca exista) $O(nm^2)$

*Cu stabilirea limitelor de interval.*

***Cautarea secventiala***

```
int caut(int v[],int val){

    for(int j=0;j<m;j++)
        if(v[j]==val) return 1;

    return 0;
}
```

***Maxim pe coloane***

```
int max_col(int col)
{
    int i, maxim = a[0][col];
    for(i=1;i<n;i++)
        if (a[i][col] > maxim)
        maxim = a[i][col];
    return maxim;
}
```

***Minim pe coloane***

```
int min_col(int col)
{
    int i, minim = a[0][col];
    for(i=1;i<n;i++)
        if (a[i][col] < minim)
        minim = a[i][col];
    return minim;
}
```

***Maximul primei coloane***

| | | | | |
|---|---|---|---|---|
| 1 | 7 | 12 | 15 | 20 |
| 3 | 4 | 7 | 11 | 16 |
| 2 | 3 | 4 | 7 | 15 |
| 7 | 8 | 9 | 10 | 11 |

***Minimul ultimei coloane***

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.6 – Element comun tuturor liniilor (daca exista) $O(nm^2)$

***Algoritmul complet de rezolvare***

***Cu stabilirea limitelor de interval.***

```
void rezolvare1()
{
   int poz_ls[10], poz_ld[10], comun, ls, ld, i, poz, val;
   ls = max_col(0); ld = min_col(m-1);
   for (i=0; i<n; i++)
   {
      poz = 0;
      while (poz < m && a[i][poz]<ls) poz++;
      if (poz == m) {cout<<"nu exista valoare comuna"; return;}
      poz_ls[i] = poz;

      poz = m - 1;
      while (poz >= 0 && a[i][poz]>ld) poz--;
      if (poz < 0) {cout<<"nu exista valoare comuna"; return;}
      poz_ld[i] = poz;
   }

      for (poz = poz_ls[0]; poz <= poz_ld[0]; poz++)
      {
         comun = 1; val = a[0][poz];
         for (i = 1; i<n; i++){

            if (caut(a[i],poz_ls[i],poz_ld[i],val) == 0){comun = 0; break;}
         }
         if (comun == 1) {cout<<"valoare comuna = "<<val<<endl; return;}
      }
   cout<<"nu exista valoare comuna"<<endl; return;
}
```

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.6 – Element comun tuturor liniilor (daca exista)

**O(nmlog m)**

***Algoritmul complet de rezolvare***

***Cu stabilirea limitelor de interval.***

```
void rezolvare2()
{
    int poz_ls[10], poz_ld[10], comun, ls, ld, i, poz, val;
    ls = max_col(0);
    ld = min_col(m-1);
    for (i=0; i<n; i++)
    {
        poz = 0;
        while (poz < m && a[i][poz]<ls) poz++;
        if (poz == m) {cout<<"nu exista valoare comuna"<<endl; return;}
        poz_ls[i] = poz;

        poz = m - 1;
        while (poz >= 0 && a[i][poz]>ld) poz--;
        if (poz < 0) {cout<<"nu exista valoare comuna"<<endl; return;}
        poz_ld[i] = poz;
    }

        for (poz = poz_ls[0]; poz <= poz_ld[0]; poz++)
        {
        comun = 1; val = a[0][poz];
        for (i = 1; i<n; i++)
            if (caut_binar(a[i],poz_ls[i],poz_ld[i],val) == -1){comun = 0; break;}
        if (comun == 1) {cout<<"valoare comuna = "<<val<<endl; return;}
        }
    cout<<"nu exista valoare comuna"; return;
}
```

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.6 – Element comun tuturor liniilor (daca exista) O(nm)

***Algoritmul complet de rezolvare***

***Sugestie de implementare:***

4. ***rezolvare3() – O(nm)*** *:*

4.1 – se considera un vector col_crt[i] = retine ultima coloana de pe linia i pana la care trebuie cautat elementul comun;

4.2 - determinam valoarea minima din col_crt - este o limita superioara pentru elementul comun;

4.3 – scadem pt fiecare linie i col_crt[i] pana ajunge pe un element <=a[lin_min][col_crt[lin_min] (mai la dreapta nu are sens sa cautam pe linie);

4.4 – definim cnt = cate limite de coloane sunt egale cu valoarea minima a[lin_min][col_crt[lin_min]];

4.5 – daca cnt = numarul de linii din matrice, atunci am gasit valoarea comuna

4.6 – daca pentru fiecare linie elementul de pe coloana col_crt[i] este =a[lin_min][col_crt[lin_min]] atunci liniile au un element comun.

4.7 – se repeta procedeul pana la iesirea din matrice sau pana la gasirea valorii comune

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.6 – Element comun tuturor liniilor (daca exista) O(nm)

***Algoritmul complet de rezolvare***

***Gasirea liniei care contine elementul minim de pe ultima coloana (elementul de la care se porneste cautarea).***

```
/*pentru fiecare linie col_crt[i] =ultima coloana de pe linia i pana la care trebuie cautat elementul comun*/
for(i = 0; i< n; i++)
    col_crt[i] = m-1;
```

| | | | | |
|---|---|---|---|---|
| 1 | 7 | 12 | 15 | 20 |
| 3 | 4 | 7 | 11 | 16 |
| 2 | 3 | 4 | 7 | 15 |
| 7 | 8 | 9 | 10 | 11 |

→

***Gasirea valorilor maxime initiale pana la care trebuie realizata cautarea pe fiecare linie***

| | | | | |
|---|---|---|---|---|
| 1 | 7 | 12 | 15 | 20 |
| 3 | 4 | 7 | 11 | 16 |
| 2 | 3 | 4 | 7 | 15 |
| 7 | 8 | 9 | 10 | 11 |

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.6 – Element comun tuturor liniilor (daca exista) O(nm)

***Algoritmul complet de rezolvare***

***Pentru val = 11 Cnt = 2. Se repeta algoritmul pe noile valori ale vectorului col_crt[]***

| | | | | |
|---|---|---|---|---|
| 1 | 7 | | | |
| 3 | 4 | 7 | 11 | |
| 2 | 3 | 4 | 7 | |
| 7 | 8 | 9 | 10 | 11 |

| | | | | |
|---|---|---|---|---|
| 1 | 7 | | | |
| 3 | 4 | 7 | | |
| 2 | 3 | 4 | 7 | |
| 7 | | | | |

```
sol = 2;
do
{
   //determinam valoarea minima din col_crt
   //este o limita superioara pentru elementul comun
   lin_min = 0;
   for(i = 0; i < n; i++)
      if(a[i][col_crt[i]] < a[lin_min][col_crt[lin_min]])
         lin_min = i;
   //determinam limitele superioare pentru fiecare linie
   cnt = 0; //=cate limite de coloane sunt egale cu valoarea minima
   for(i = 0; i < n; i++)
     while(a[i][col_crt[i]] > a[lin_min][col_crt[lin_min]])
         if(col_crt[i] == 0)
         {
            sol = 0;i=n; break;
         }
         else col_crt[i]--;

    for(i = 0; i < n; i++)
      if(a[lin_min][col_crt[lin_min]]==a[i][col_crt[i]] )
         cnt++;
   /*daca pentru fiecare linie elementul de pe coloana col_crt[i] este
   a[lin_min][col_crt[lin_min]] atunci liniile au un element comun*/
   if(cnt == n)
      sol = 1;
}while(sol == 2);
```

# 2. Tablouri bidimensionale – Aplicatii

## Aplicatie 2.6 – Element comun tuturor liniilor (daca exista) O(nm)

***Algoritmul complet de rezolvare***

```
if(sol == 0)
    cout << "nu exista valoare comuna"<< endl;
else
    cout << "valoare comuna = " << a[lin_min][col_crt[lin_min]] << endl;
```

***Pentru val = 7 Cnt = 4 – Numarul de aparitii ale valorii = numarul de linii din matrice (cautare cu succes) .***

| | | | | |
|---|---|---|---|---|
| 1 | 7 | | | |
| 3 | 4 | 7 | | |
| 2 | 3 | 4 | 7 | |
| 7 | | | | |

# Operatii pe tablouri bidimensionale

## Concluzii

S-au recapitulat notiunile urmatoare:

- Tablouri unidimensionale;
- Tablouri bidimensionale – notiuni teoretice;
- Tablouri bidimensionale - Aplicatii.

**Succes!**