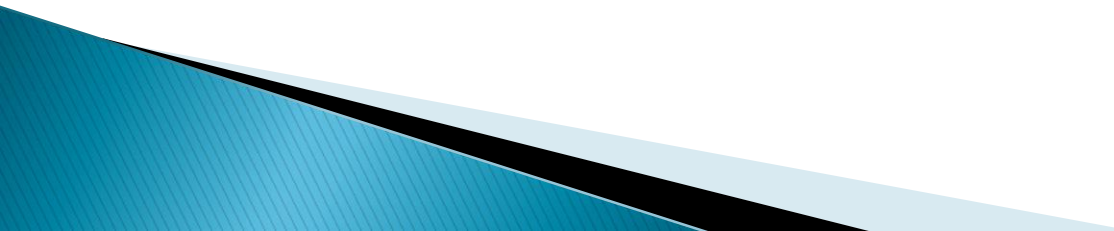


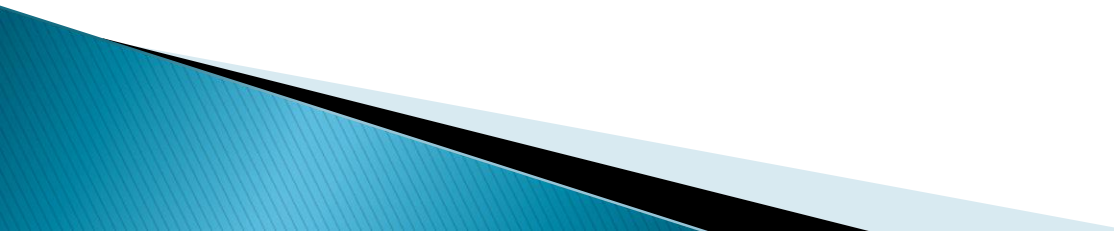
Metoda Backtracking



Exemple

- ▶ Permutări, combinări, aranjamente
 - ▶ Colorarea hărților
 - ▶ Toate subșirurile crescătoare de lungime maximă
 - ▶ Partițiile unui număr n
 - ▶ Labirint
- 

Permutări

- ▶ **Reprezentarea soluției**
 - ▶ **Condiții interne (finale)**
 - ▶ **Condiții de continuare (!!pentru x_k)**
- 

Permutări

- ▶ **Reprezentarea soluției**

$\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$, unde

$\mathbf{x}_k \in \{1, 2, \dots, n\} \quad (p_k = 1, u_k = n) .$

- ▶ **Condiții interne (finale)**

- ▶ **Condiții de continuare (!!pentru \mathbf{x}_k)**

Permutări

► Reprezentarea soluției

$\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$, unde

$\mathbf{x}_k \in \{1, 2, \dots, n\} \quad (p_k = 1, u_k = n) .$

► Condiții interne (finale)

$\mathbf{x}_i \neq \mathbf{x}_j$ pentru orice $i \neq j$.

► Condiții de continuare (!!pentru \mathbf{x}_k)

Permutări

► Reprezentarea soluției

$\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$, unde

$\mathbf{x}_k \in \{1, 2, \dots, n\} \quad (p_k = 1, u_k = n) .$

► Condiții interne (finale)

$\mathbf{x}_i \neq \mathbf{x}_j$ pentru orice $i \neq j$.

► Condiții de continuare (!!pentru \mathbf{x}_k)

$\mathbf{x}_i \neq \mathbf{x}_k$ pentru orice $i \in \{1, 2, \dots, \mathbf{k}-1\}$

Permutări, $n=3$



1

1 1

1 2

1 2 1

1 2 2

1 2 3

1 2 3 soluție

1 3

1 3 1

1 3 2

1 3 2 soluție

1 3 3

2

2 1

2 1 1

2 1 2

2 1 3

2 1 3 soluție

etc

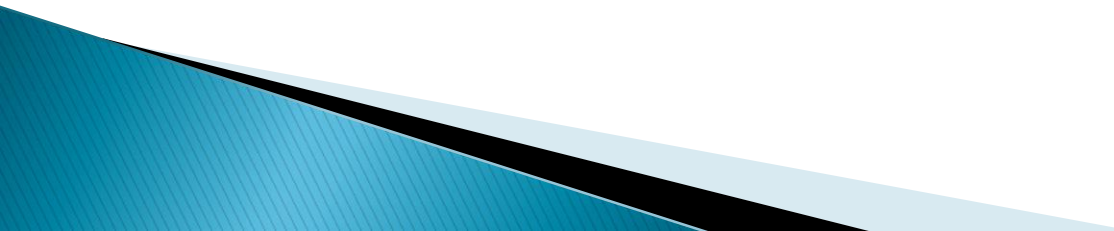
Colorarea hărților



► Se consideră o hartă cu n țări.

Se cere colorarea ei folosind cel mult 4 culori, astfel încât oricare două țări vecine să fie colorate diferit

Colorarea hărților

- ▶ **Reprezentarea soluției**
 - ▶ **Condiții interne (finale)**
 - ▶ **Condiții de continuare (!!pentru x_k)**
- 

Colorarea hărților

► Reprezentarea soluției

$\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$, unde

\mathbf{x}_k = culoarea cu care este colorată țara k

$\mathbf{x}_k \in \{1, 2, 3, 4\}$ ($p_k = 1, u_k = 4$).

► Condiții interne (finale)

► Condiții de continuare (!!pentru \mathbf{x}_k)

Colorarea hărților

► Reprezentarea soluției

$\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$, unde

\mathbf{x}_k = culoarea cu care este colorată țara k

$\mathbf{x}_k \in \{1, 2, 3, 4\}$ ($p_k = 1, u_k = 4$).

► Condiții interne (finale)

$\mathbf{x}_i \neq \mathbf{x}_j$ pentru orice două țări **vecine** i și j .

► Condiții de continuare (!!pentru \mathbf{x}_k)

Colorarea hărților

► Reprezentarea soluției

$\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$, unde

\mathbf{x}_k = culoarea cu care este colorată țara k

$\mathbf{x}_k \in \{1, 2, 3, 4\}$ ($p_k = 1, u_k = 4$).

► Condiții interne (finale)

$\mathbf{x}_i \neq \mathbf{x}_j$ pentru orice două țări **vecine** i și j .

► Condiții de continuare (!!pentru \mathbf{x}_k)

$\mathbf{x}_i \neq \mathbf{x}_k$ pentru orice țară $i \in \{1, 2, \dots, k-1\}$
vecină cu țara k

Implementare – varianta recursivă

```
boolean cont(int k){  
    for(int i=1;i<k;i++)  
        if(a[i][k]==1 && x[i]==x[k])  
            return false;  
    return true;  
}
```

Implementare – varianta recursivă

```
boolean cont(int k){  
    for(int i=1;i<k;i++)  
        if(a[i][k]==1 && x[i]==x[k])  
            return false;  
    return true;  
}
```

```
void backrec(int k){  
    if(k==n+1)  
        retsol(x);  
  
}
```

Implementare – varianta recursivă

```
boolean cont(int k){
    for(int i=1;i<k;i++)
        if(a[i][k]==1 && x[i]==x[k])
            return false;
    return true;
}

void backrec(int k){
    if(k==n+1)
        retsol(x);
    else
        for(int i=1;i<=4;i++){
            x[k]=i;                //atribuie
            if (cont(k))           //avanseaza
                backrec(k+1);
        }
}
```

Implementare – varianta nerecursivă

```
void back() {
    int k=1;
    x=new int[n+1];
    for(int i=1;i<=n;i++) x[i]=0;
    while(k>0) {
        if(k==n+1) {retsol(x); k--;} //revenire dupa sol

    }
}
```


Implementare – varianta nerecursivă

```
void back() {  
    int k=1;  
    x=new int[n+1];  
    for(int i=1;i<=n;i++) x[i]=0;  
    while(k>0) {  
        if(k==n+1) {retsol(x); k--;} //revenire dupa sol  
        else{  
            if(x[k]<4) {  
                x[k]++; //atribuie  
                if (cont(k)) k++; //si avanseaza  
            }  
        }  
    }  
}
```

Implementare – varianta nerecursivă

```
void back() {  
    int k=1;  
    x=new int[n+1];  
    for(int i=1;i<=n;i++) x[i]=0;  
    while(k>0) {  
        if(k==n+1) { retsol(x); k--;} //revenire dupa sol  
        else{  
            if(x[k]<4) {  
                x[k]++; //atribuie  
                if (cont(k)) k++; //si avanseaza  
            }  
            else{ x[k]=0; k--; } //revenire  
        }  
    }  
}
```

Subșiruri crescătoare lungime maximă



- ▶ Fie vectorul $a=(a_1, \dots, a_n)$. Să se determine toate subșirurile crescătoare de lungime maximă.

Subșiruri crescătoare lungime maximă

- ▶ Fie vectorul $a=(a_1,\dots,a_n)$. Să se determine toate subșirurile crescătoare de lungime maximă.
- ▶ **Subproblemă:**
 $\text{lung}[i]$ = lungimea maximă a unui subșir crescător ce începe pe poziția i
- ▶ **Soluție problemă:**
$$\text{lmax} = \max\{\text{lung}[i] \mid i = 1, 2, \dots, n\}$$

Subșiruri crescătoare lungime maximă

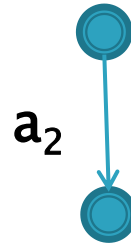
- ▶ Un subșir crescător de lungime maximă începe cu o poziție i cu $lung[i] = l_{max}$
- ▶ Următorul element după a_i este un a_j cu proprietățile

Subșiruri crescătoare lungime maximă

- ▶ Un subșir crescător de lungime maximă începe cu o poziție i cu $lung[i] = l_{max}$
- ▶ Următorul element după a_i este un a_j cu proprietățile
 - $i < j$
 - $a_j > a_i$
 - $lung[j] = lung[i] - 1$ (v.PD – numărarea subșirurilor)
= **acei a_j posibili succesori ai lui i , pentru care se realizează egalitate în relația de recurență** (în PD se reține în $succ[i]$ un singur astfel de j)

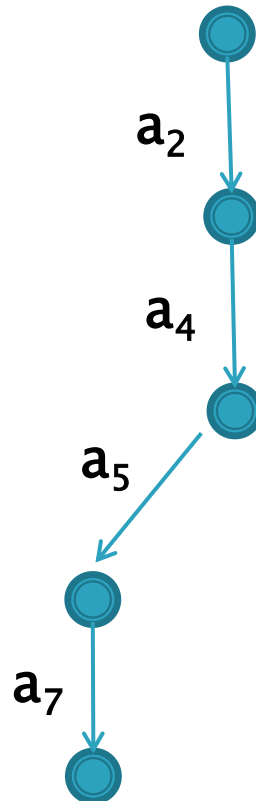
Subșiruri crescătoare lungime maximă

a:	8	1	7	4	6	5	11
	¹	²	³	⁴	⁵	⁶	⁷
lung :	2	4	2	3	2	2	1



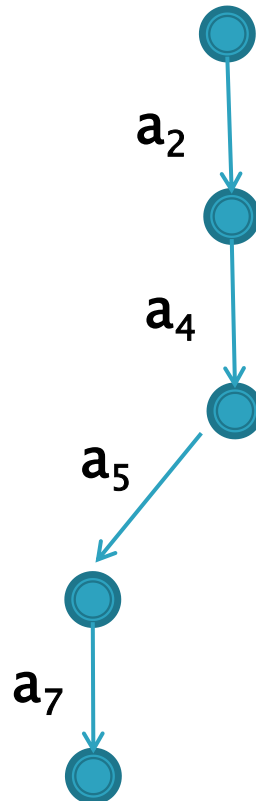
Subșiruri crescătoare lungime maximă

a:	8	1	7	4	6	5	11
	¹	²	³	⁴	⁵	⁶	⁷
lung :	2	4	2	3	2	2	1



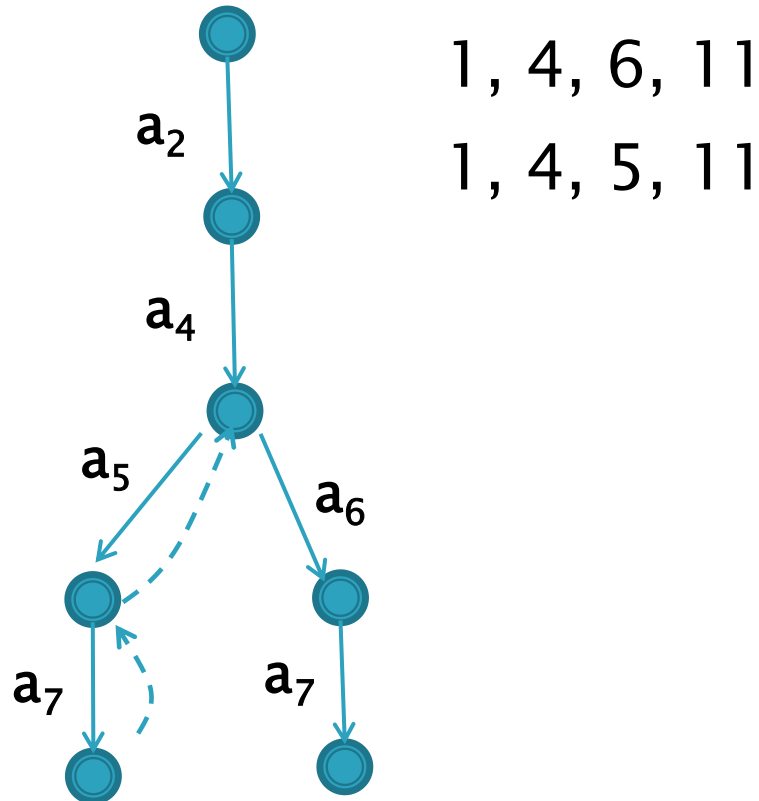
Subșiruri crescătoare lungime maximă

a:	8	1	7	4	6	5	11
	¹	²	³	⁴	⁵	⁶	⁷
lung :	2	4	2	3	2	2	1

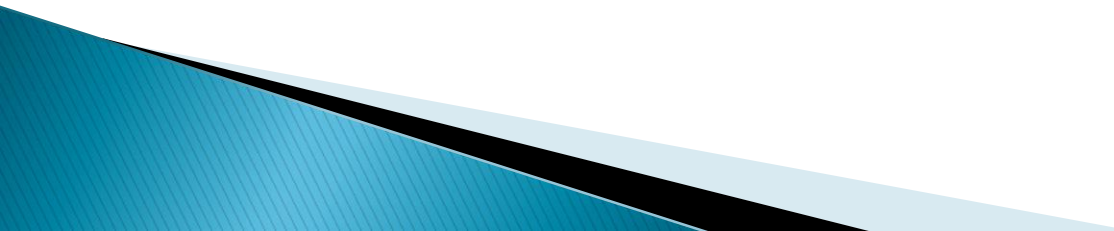


Subșiruri crescătoare lungime maximă

a:	8	1	7	4	6	5	11
	¹	²	³	⁴	⁵	⁶	⁷
lung :	2	4	2	3	2	2	1



Subșiruri crescătoare lungime maximă

- ▶ **Reprezentarea soluției**
 - ▶ **Condiții interne (finale)**
 - ▶ **Condiții de continuare**
- 

Subșiruri crescătoare lungime maximă

- ▶ **Reprezentarea soluției**

$\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_{l_{\max}}\}$, unde

$\mathbf{x}_k \in \{1, \dots, n\}$ poziție din vectorul \mathbf{a}

- ▶ **Condiții interne (finale)**

$$a_{x_1} < a_{x_2} < \dots < a_{x_{l_{\max}}}$$

- ▶ **Condiții de continuare**

Subșiruri crescătoare lungime maximă

► Reprezentarea soluției

$\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_{l_{\max}}\}$, unde

$\mathbf{x}_k \in \{1, \dots, n\}$ poziție din vectorul a

► Condiții interne (finale)

$$a_{\mathbf{x}_1} < a_{\mathbf{x}_2} < \dots < a_{\mathbf{x}_{l_{\max}}}$$

► Condiții de continuare

$$\text{lung}[a[\mathbf{x}_1]] = l_{\max}$$

$$\mathbf{x}_{k-1} < \mathbf{x}_k, \quad a_{\mathbf{x}_{k-1}} < a_{\mathbf{x}_k}, \quad \text{lung}[\mathbf{x}_k] = \text{lung}[\mathbf{x}_{k-1}] - 1$$

```

void scribe(int k){
    if (k==lmax+1){    //retsol();
        for (int i=1;i<=lmax;i++)
            System.out.print(a[x[i]]+" ");
        System.out.println();
    }
    else
        for (int j=x[k-1]+1;j<=n;j++){
            x[k]=j;

        }
}

```

```

for(int i=1;i<=n;i++)
    if (lung[i]==lmax) {
        x[1]=i;
        scribe(2);
    }

```

```

void scribe(int k){
    if (k==lmax+1){    //retsol();
        for (int i=1;i<=lmax;i++)
            System.out.print(a[x[i]]+" ");
        System.out.println();
    }
    else
        for (int j=x[k-1]+1;j<=n;j++){
            x[k]=j;
            if ((a[x[k-1]]<a[x[k]]) &&
                (lung[x[k-1]]==1+lung[x[k]]))
                scribe(k+1);
        }
}

```

```

for(int i=1;i<=n;i++)
    if (lung[i]==lmax) {
        x[1]=i;
        scribe(2);
    }

```

Metoda Backtracking – Partiții

- ▶ **Reprezentarea soluției**

$\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k\}$, unde

$\mathbf{x}_i \in \{1, \dots, n\}$

- ▶ **Condiții interne (finale)**

- ▶ **Condiții de continuare**

Metoda Backtracking – Partiții

► Reprezentarea soluției

$$\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k\}, \text{ unde}$$
$$\mathbf{x}_i \in \{1, \dots, n\}$$

► Condiții interne (finale)

$$\mathbf{x}_1 + \mathbf{x}_2 + \dots + \mathbf{x}_k = n$$

Pentru unicitate: $\mathbf{x}_1 \leq \mathbf{x}_2 \leq \dots \leq \mathbf{x}_k$

► Condiții de continuare

Metoda Backtracking – Partiții

► Reprezentarea soluției

$$\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k\}, \text{ unde} \\ \mathbf{x}_i \in \{1, \dots, n\}$$

► Condiții interne (finale)

$$\mathbf{x}_1 + \mathbf{x}_2 + \dots + \mathbf{x}_k = n$$

Pentru unicitate: $\mathbf{x}_1 \leq \mathbf{x}_2 \leq \dots \leq \mathbf{x}_k$

► Condiții de continuare

$$\mathbf{x}_{k-1} \leq \mathbf{x}_k \longrightarrow \mathbf{x}_k \in \{\mathbf{x}_{k-1}, \dots, n\} = \mathbf{X}_k$$

$$\mathbf{x}_1 + \mathbf{x}_2 + \dots + \mathbf{x}_k \leq n$$

Implementare – varianta recursivă

```
void backrec(int k) {  
    if(s==n) { //este solutie  
        retsol(x, k-1);  
        return;  
    }  
}
```

```
}
```

```

void backrec(int k) {
    if(s==n) { //este solutie
        retsol(x, k-1);
        return;
    }
    for(int i=x[k-1]; i<=n; i++) {
        x[k]=i;
        if (s+x[k]<=n) { //cont
            s+=x[k] ;
            backrec(k+1);
            s-=x[k] ;
        }
        else
            return;
    }
}

```

```
void retsol(int[] x,int k){  
    for(int i=1;i<=k;i++)  
        System.out.print(x[i]+" ");  
    System.out.println();  
}
```

```
void backrec(){  
    x=new int[n+1];  
    x[0]=1; //prima valoare pentru x[1]  
    s=0;  
    backrec(1);  
}
```

Implementare – varianta nerecursivă

```
void back() {
    int k=1, s=0; int x[]=new int[n+1];
    x[1]=0;
    while(k>=1) {
        if(x[k]<n) {
            x[k]++; s++;
            if(s<=n) {//cont - verific. conditiilor de cont

```



```

void back() {
    int k=1, s=0; int x[]=new int[n+1];
    x[1]=0;
    while(k>=1) {
        if(x[k]<n) {
            x[k]++; s++;
            if(s<=n) { // cont - verific. conditiilor de cont
                if(s==n) { // dc este sol
                    retsol(x, k);
                    s=s-x[k]; k--; // revenire dupa sol
                }
            }
        }
    }
}

```

```

void back() {
    int k=1, s=0; int x[]=new int[n+1];
    x[1]=0;
    while(k>=1) {
        if(x[k]<n) {
            x[k]++; s++;
            if(s<=n) { //cont - verific. conditiilor de cont
                if(s==n) { //dc este sol
                    retsol(x, k);
                    s=s-x[k]; k--; //revenire dupa sol
                }
            }
            else{ k++; x[k]=x[k-1]-1; s+=x[k]; //avansare
            }
        }
    }
}

```

```

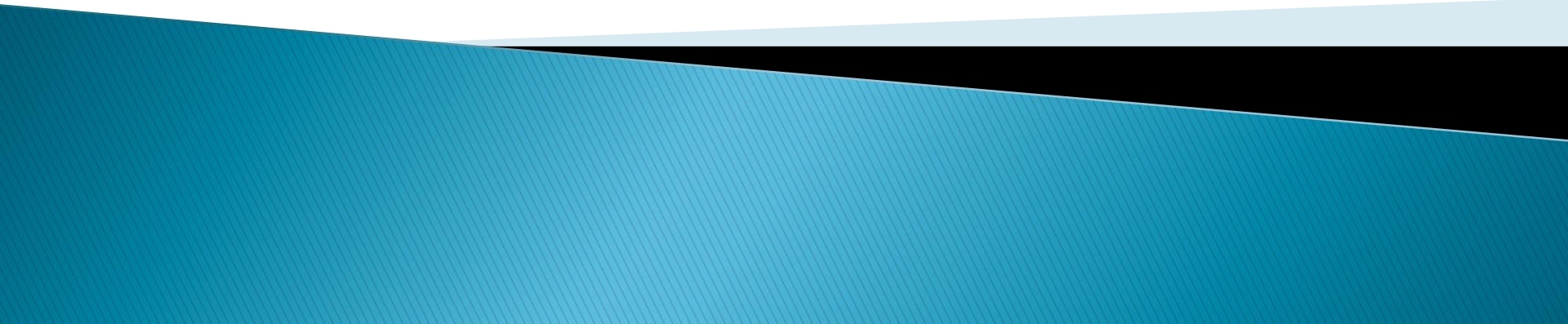
void back() {
    int k=1, s=0; int x[]=new int[n+1];
    x[1]=0;
    while(k>=1) {
        if(x[k]<n) {
            x[k]++; s++;
            if(s<=n) { //cont - verific. conditiilor de cont
                if(s==n) { //dc este sol
                    retsol(x, k);
                    s=s-x[k]; k--; //revenire dupa sol
                }
            }
            else{ k++; x[k]=x[k-1]-1; s+=x[k]; //avansare
            }
        }
        else{ s=s-x[k]; k--; //revenire
        }
    }
}

```

Metoda Backtracking – Partiții

- ▶ **DE EVITAT** recalcularea lui s la fiecare pas ca fiind
$$s = x[1] + \dots + x[k]$$

Backtracking în plan



Backtracking în plan

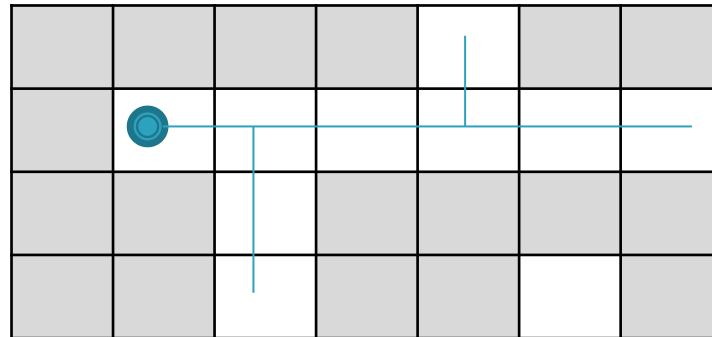
- ▶ **Labirint.** Se consideră un caroiaj (matrice) A cu m linii și n coloane. Pozițiile pot fi:

- libere: $a_{ij}=0$;
- ocupate: $a_{ij}=1$.

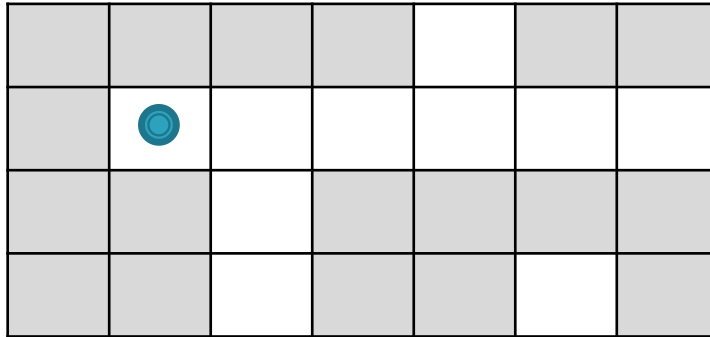
Se mai dă o poziție (i_0, j_0) . Se caută **toate** drumurile care ies în afara matricei, trecând numai prin poziții libere (fără a trece de două ori prin aceeași poziție).

Variante:

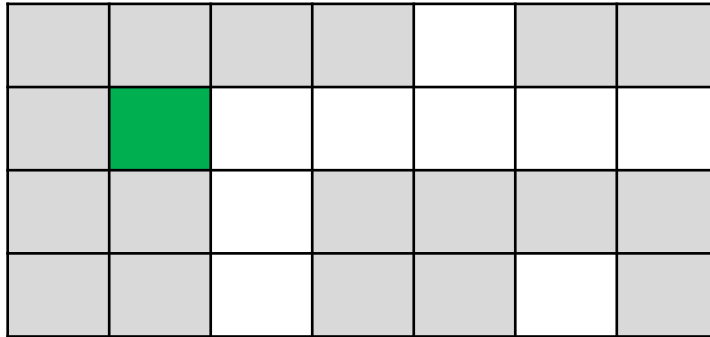
- drumul maxim
- drumul minim



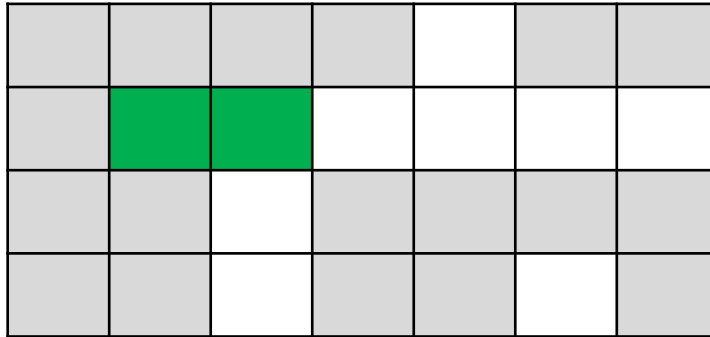
Backtracking în plan



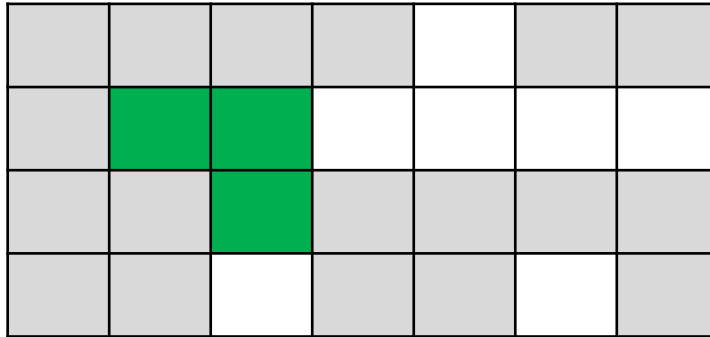
Backtracking in plan



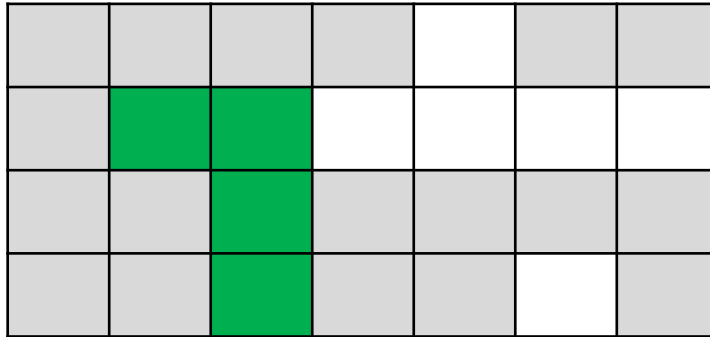
Backtracking în plan



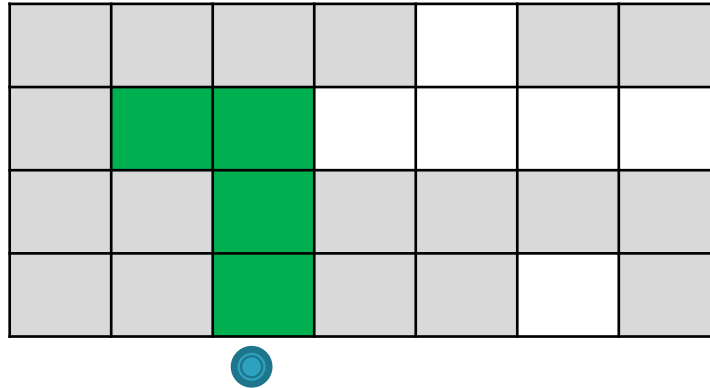
Backtracking în plan



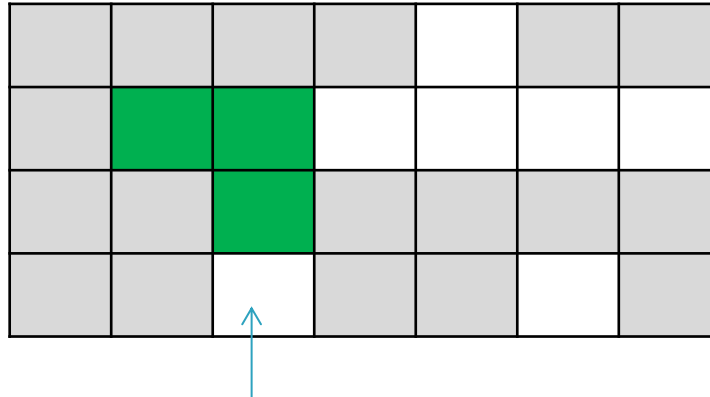
Backtracking în plan



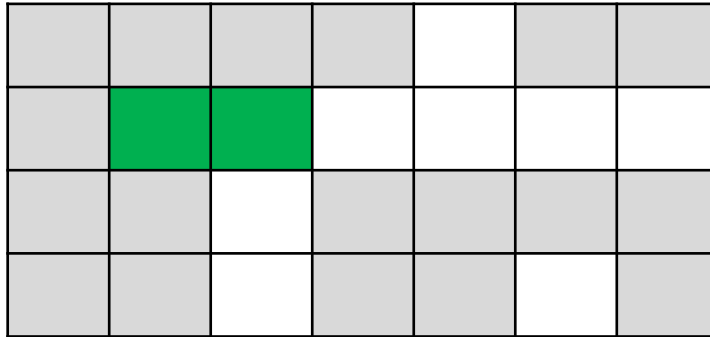
Backtracking in plan



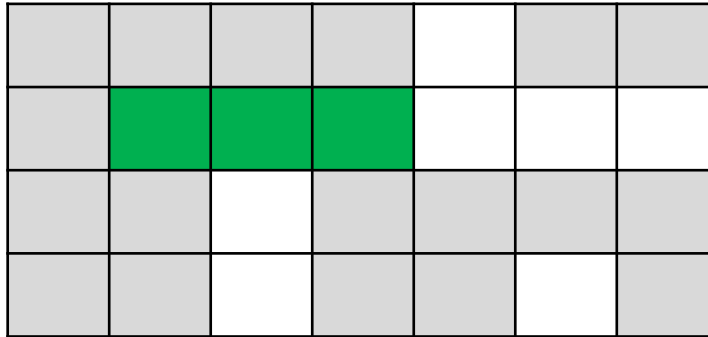
Backtracking in plan



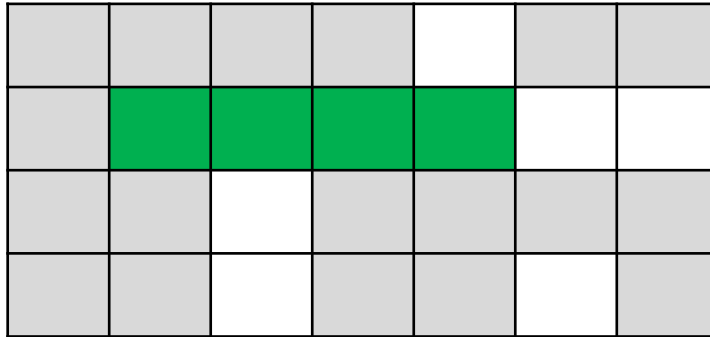
Backtracking în plan



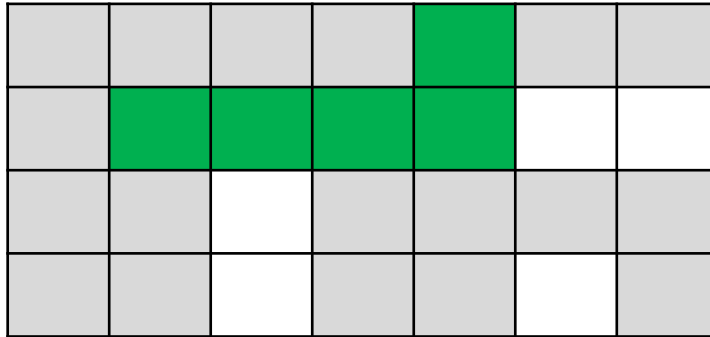
Backtracking în plan



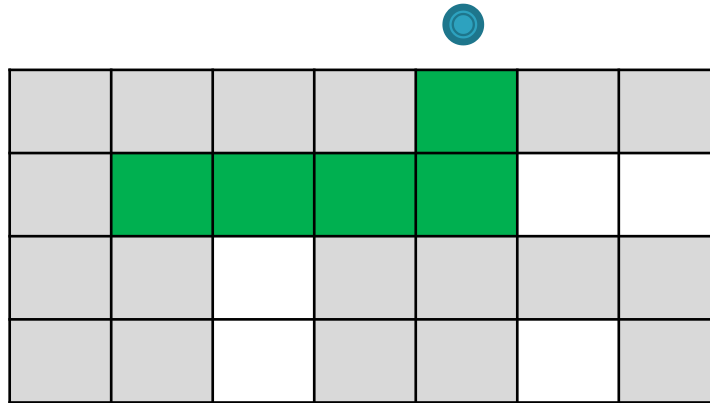
Backtracking în plan



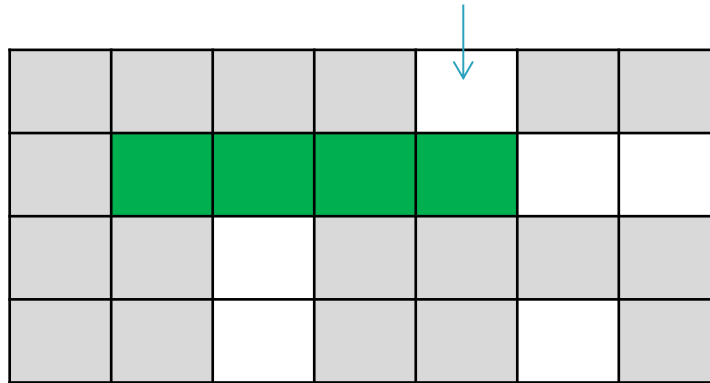
Backtracking în plan



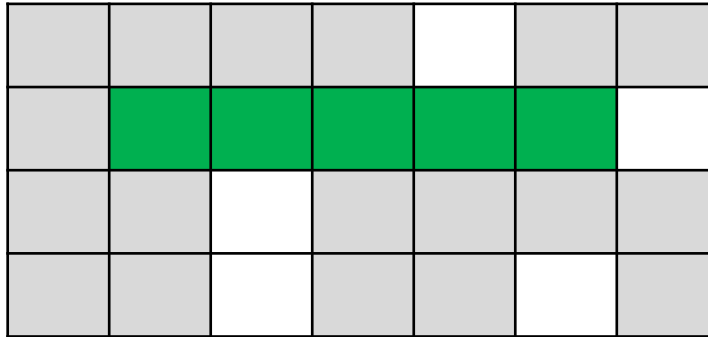
Backtracking în plan



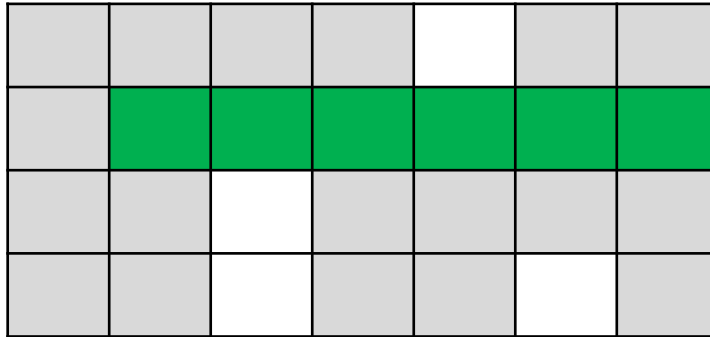
Backtracking in plan



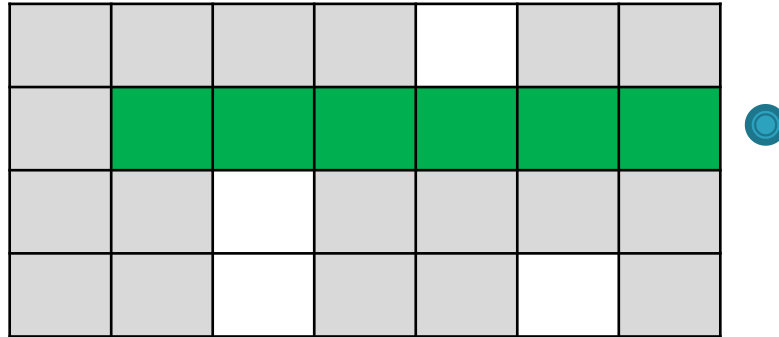
Backtracking în plan



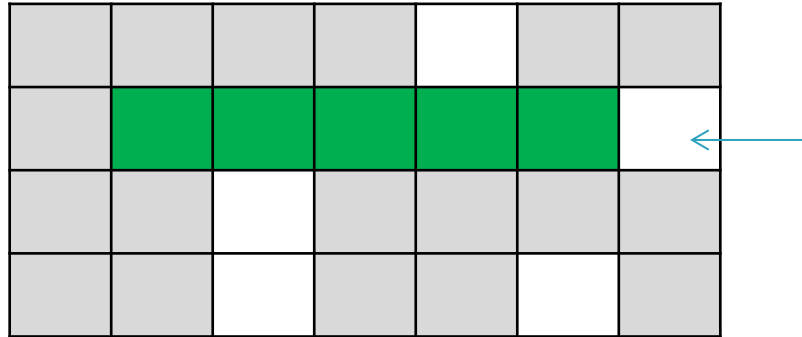
Backtracking în plan



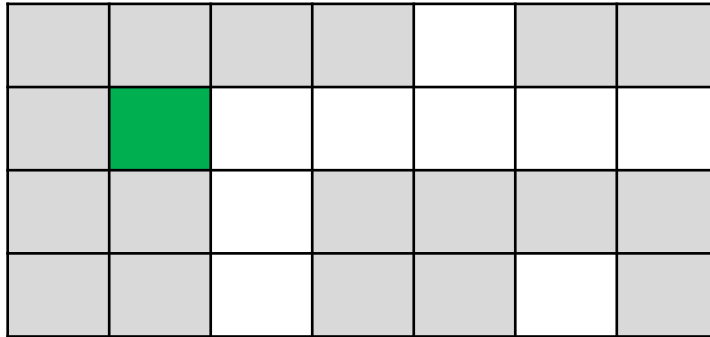
Backtracking in plan



Backtracking in plan



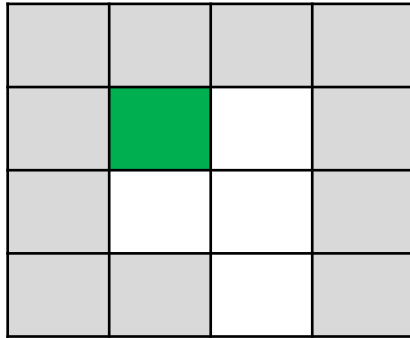
Backtracking in plan



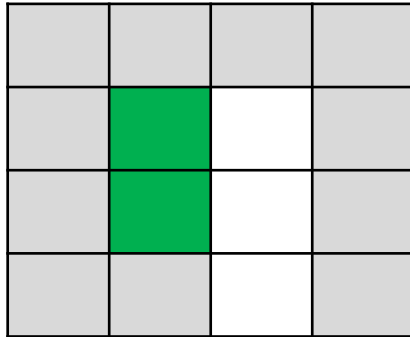
Backtracking în plan

- ▶ **Observație:** este important să demarcăm celulele când facem pasul înapoi

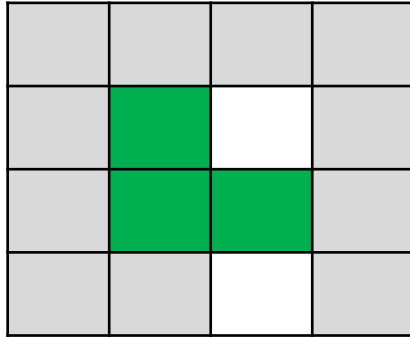
Backtracking in plan



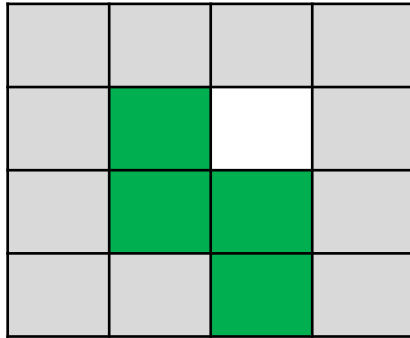
Backtracking în plan



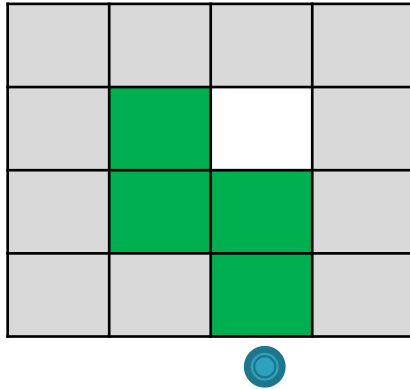
Backtracking in plan



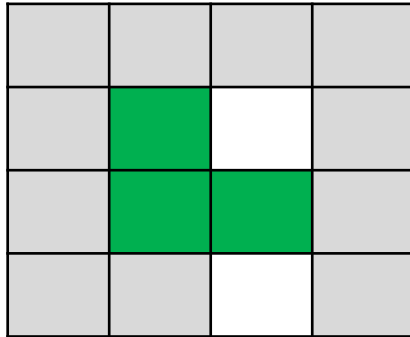
Backtracking in plan



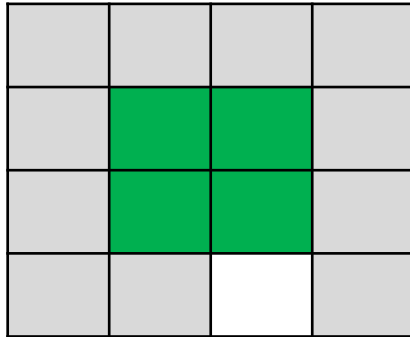
Backtracking în plan



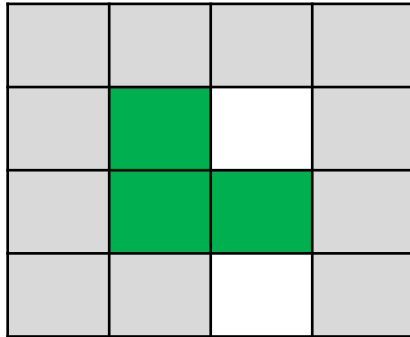
Backtracking in plan



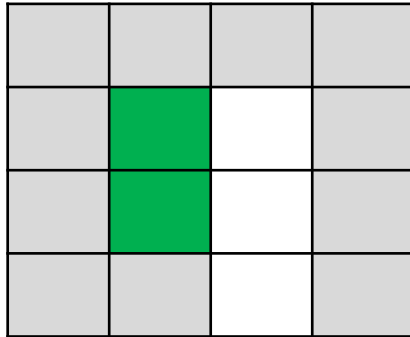
Backtracking in plan



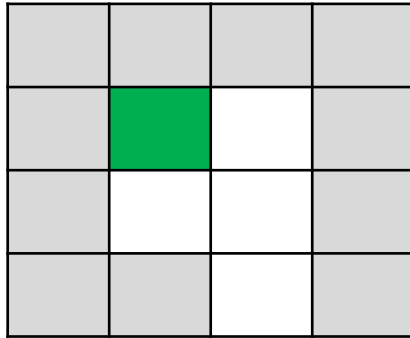
Backtracking în plan



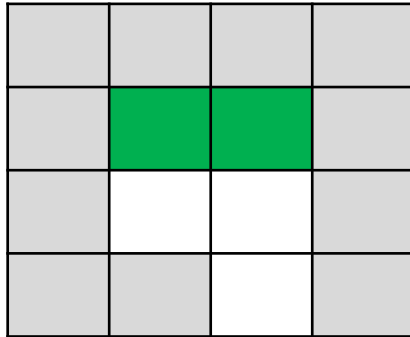
Backtracking în plan



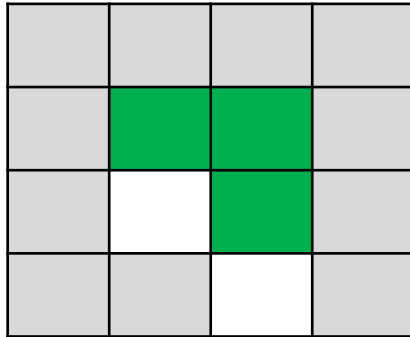
Backtracking in plan



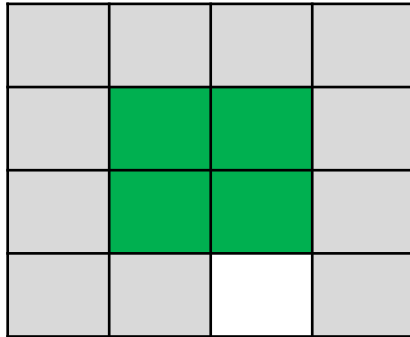
Backtracking in plan



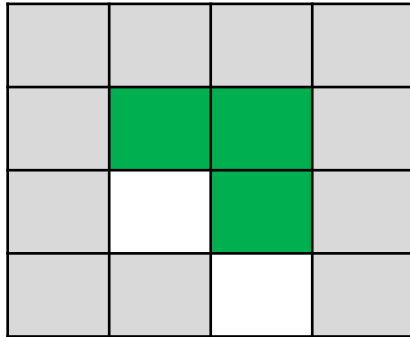
Backtracking in plan



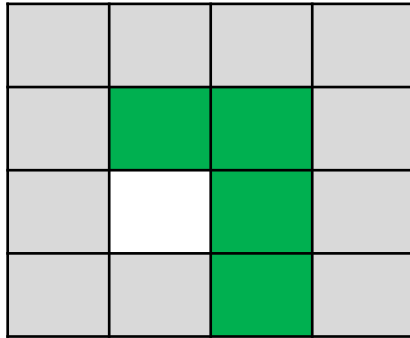
Backtracking in plan



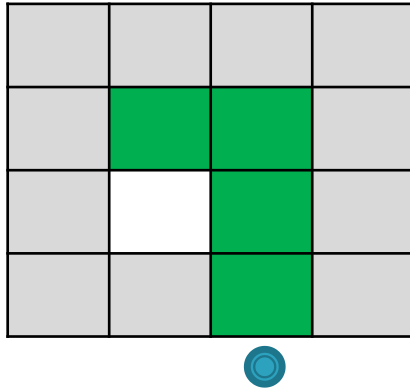
Backtracking in plan



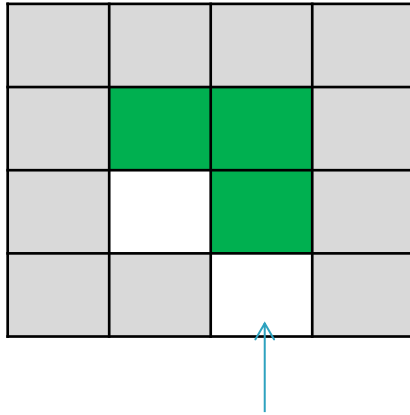
Backtracking in plan



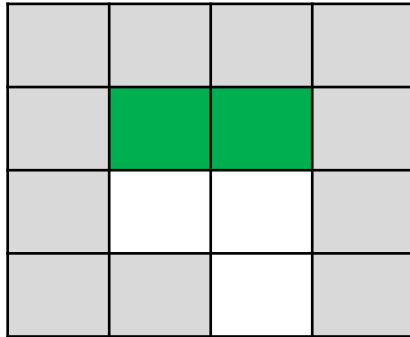
Backtracking in plan



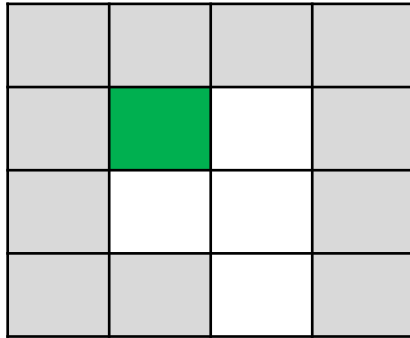
Backtracking in plan



Backtracking in plan



Backtracking in plan



Backtracking în plan

- ▶ Bordăm matricea cu 2 pentru a nu studia separat ieșirea din matrice.

- ▶ **Reprezentarea soluției**

$\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k\}$, unde
 \mathbf{x}_i = a i-a celulă din drum

- ▶ **Condiții interne (finale)**

- ▶ **Condiții de continuare**

► Reprezentarea soluției

$\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k\}$, unde
 \mathbf{x}_i = a i-a celulă din drum

► Condiții interne (finale)

\mathbf{x}_k = celulă din afara matricei (marcată cu 2)

$\{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_{k-1}\}$ – celule libere (marcată cu 0)

► Condiții de continuare

\mathbf{x}_k celulă liberă prin care nu am mai trecut

Backtracking în plan

- ▶ dacă poziția este liberă și putem continua, setăm $a_{ij} = -1$ (a fost atinsă), continuăm
- ▶ repunem $a_{ij} = 0$ la întoarcere (din recursivitate)

Backtracking în plan

- ▶ dacă poziția este liberă și putem continua, setăm $a_{ij} = -1$ (a fost atinsă), continuăm
- ▶ repunem $a_{ij} = 0$ la întoarcere (din recursivitate)
- ▶ Matricea deplasărilor $dep1$ cu două linii și $ndep1$ coloane :

$$\begin{pmatrix} 1 & 0 & -1 & 0 \\ 0 & -1 & 0 & 1 \end{pmatrix}$$

```
void back(i, j){
    for (t = 1; t<=ndepl; t++){
        ii = i + depl[1][t]
        jj = j + depl[2][t];

    }
}
```

```

void back (i, j) {
    for (t = 1; t<=ndepl; t++) {
        ii = i + depl[1][t]
        jj = j + depl[2][t];
        if (a[ii][jj] == 1)
        else
            if (a[ii][jj] == 2)
                retsol(x,k);
            else
                if (a[ii][jj] == 0) {

                                }

        }
    }
}

```

```

void back (i, j) {
    for (t = 1; t<=ndepl; t++) {
        ii = i + depl[1][t]
        jj = j + depl[2][t];
        if (a[ii][jj] == 1)
        else
            if (a[ii][jj] == 2)
                retsol(x,k);
            else
                if (a[ii][jj] == 0) {
                    k = k+1;           //creste
                     $x_k \leftarrow (ii, jj);$ 
                    a[i][j] = -1; //marcam
                    back(ii, jj);
                }
    }
}

```

```

void back (i, j) {
    for (t = 1; t<=ndepl; t++) {
        ii = i + depl[1][t]
        jj = j + depl[2][t];
        if (a[ii][jj] == 1)
        else
            if (a[ii][jj] == 2)
                retsol(x,k);
            else
                if (a[ii][jj] == 0) {
                    k = k+1;           //creste
                     $x_k \leftarrow (ii, jj);$ 
                    a[i][j] = -1; //marcam
                    back(ii, jj);
                    a[i][j] = 0;  //demarcam
                    k = k-1 ;      //scade
                }
    }
}

```

Apel:

$x_1 \leftarrow (i_0, j_0);$

$k = 1;$

back(i_0, j_0)

Backtracking în plan

- **Cuvinte.** Se consideră un carioaj (matrice) A cu m linii și n coloane cu litere și un cuvânt c.

Să se determine dacă c se poate regăsi în matrice pornind dintr-o celulă și deplasându-ne în oricare din celulele vecine pe orizontală, verticală sau diagonală fără a trece de două ori prin aceeași celulă – **TEMĂ**

c = test

s	s	e	s	t	t	a
b	a	s	e	e	e	t
b	t	e	e	t	e	e
a	a	a	e	c	e	e

s	s	e	s	t	t	a
b	a	s	e	e	e	t
b	t	e	e	t	e	e
a	a	a	e	c	e	e

Backtracking în plan

► Cuvinte

Indicații:

- similar cu Labirint
- punct de start poate fi orice celulă care conține prima literă din c
- printre condiții de continuare: litera la care am ajuns în matrice la pasul k trebuie să fie a k -a literă din c