

# Problema monedelor



- ▶ Avem la dispoziție un număr nelimitat de monede de valori  $\{v_1, v_2, \dots, v_n\}$  și o sumă  $S$  care trebuie plătită.

Să se determine o modalitate de plată a sumei  $S$  folosind un număr minim de monede (știind că este posibil să plătim suma  $S$ ).

**Exemplu:** Pentru monedele  $\{6, 3, 5, 1\}$  și  $S=8$ , plata optimă este  $5 + 3$

# Problema monedelor

## ▶ Principiu de optimalitate

Dacă ultima monedă pe care o folosim pentru plata optimă a unei sume  $s$  este  $v_i$ , atunci restul monedelor folosite pentru această plată optimă constituie o soluție optimă pentru  $s - v_i$ .

## ▶ Subproblemă:

$nr[s]$  = numărul minim de monede necesare pentru a plăti o sumă  $s \leq S$ .

## ▶ Soluție $nr[S]$

# Problema monedelor

- ▶ **Știm direct**

$$\text{nr}[0] = 0$$

- ▶ **Relație de recurență**

$$\text{nr}[s] = \min\{1 + \text{nr}[s - v_i], 1 \leq i \leq n, v_i \leq s\}$$

- ▶ **Ordinea de calcul**

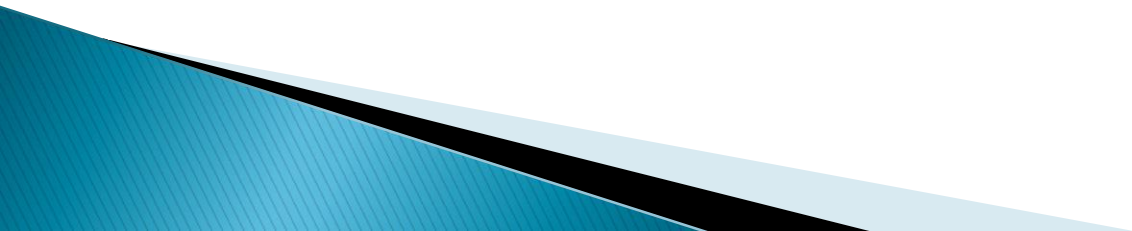
$$s = 1, \dots, S \rightarrow \text{Complexitate } O(nS)$$

- ▶ **Memorarea unei soluții**

$\text{moneda}[s]$  = indicele  $i$  pentru care se realizează  
minimul din formula pentru  $\text{nr}[s]$

# Problema monedelor

```
nr[0] = 0; moneda[0] = -1;
```



# Problema monedelor

```
nr[0] = 0; moneda[0] = -1;  
for (s = 1; s ≤ S; s++) {  
    nr[s] = ∞; moneda[s] = -1;
```

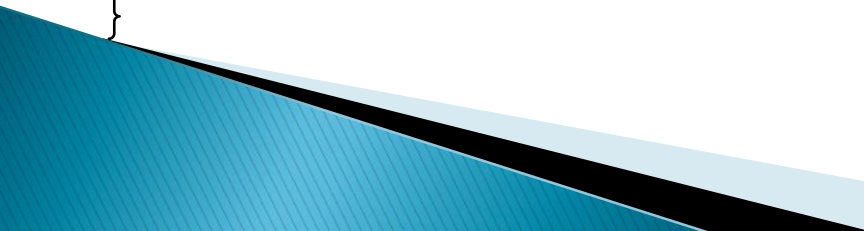
# Problema monedelor

```
nr[0] = 0; moneda[0] = -1;
for (s = 1; s<=S; s++) {
    nr[s] = ∞; moneda[s] = -1;
    for (i=1; i<=n; i++)
        if ( $v_i \leq s$  &&  $nr[s - v_i] + 1 < nr[s]$ ) {
            nr[s] = nr[s -  $v_i$ ] + 1;
            moneda[s] = i;
        }
}
scrie nr[S]
```

# Problema monedelor

```
nr[0] = 0; moneda[0] = -1;
for (s = 1; s<=S; s++) {
    nr[s] = ∞; moneda[s] = -1;
    for (i=1; i<=n; i++)
        if ( $v_i \leq s$  &&  $nr[s - v_i] + 1 < nr[s]$ ) {
            nr[s] = nr[s -  $v_i$ ] + 1;
            moneda[s] = i;
        }
}

scrie nr[S]
s = S
for (i = 1; i<= nr[S]; i++) {
    scrie v[moneda[s]]
    s = s - v[ moneda[s]]
}
```



# Problema monedelor

## ► Temă

Dacă valorile monedelor satisfac proprietățile:

$$v_1 \geq v_2 \geq \dots \geq v_n = 1$$

și

$$v_i \mid v_{i-1} \text{ pentru orice } i \in \{2, 3, \dots, n\}$$

strategia Greedy furnizează soluția optimă.



# Descompunerea unui dreptunghi în pătrate



► Se consideră un dreptunghi cu laturile de  $m$ , respectiv  $n$  unități ( $m < n$ ). Asupra sa se pot face tăieturi *complete* pe orizontală sau verticală.

Se cere numărul minim de pătrate (cu laturi numere întregi) în care poate fi descompus dreptunghiul.

**Exemplu .** Un dreptunghi  $5 \times 6$  poate fi descompus în două pătrate de latură 3 și 3 pătrate de latură 2.

# Descompunerea unui dreptunghi în pătrate

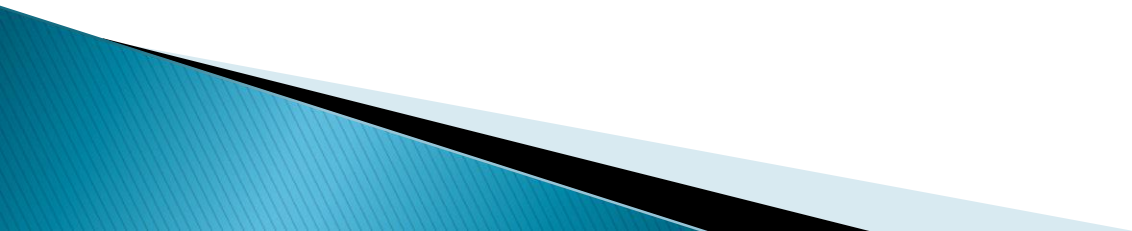
- ▶ **Principiu de optimalitate**

- ▶ **Subproblemă:**

$a[i][j]$  = numărul minim de pătrate în care poate fi descompus un dreptunghi de laturi  $i$  și  $j$

- ▶ **Soluție**  $a[m][n]$

# Descompunerea unui dreptunghi în pătrate

- ▶ Știm direct
  - ▶ Relație de recurență
  - ▶ Ordinea de parcurgere a grafului de dependențe (ordinea de calcul)
- 

# Descompunerea unui dreptunghi în pătrate

## ► Știm direct

$$a[i][1] = i, \quad \forall i = 1, \dots, m$$

$$a[1][j] = j, \quad \forall j = 1, \dots, n$$

$$a[i][i] = 1, \quad \forall i = 1, \dots, m$$

## ► Relație de recurență

$$a[i][j] = \min\{\alpha, \beta\}, \quad \text{unde}$$

$$\alpha = \min\{a[i][k] + a[i][j-k] \mid k \leq \lfloor j/2 \rfloor\}$$

$$\beta = \min\{a[k][j] + a[i-k][j] \mid k \leq \lfloor i/2 \rfloor\}$$

$$a[j][i] = a[i][j], \quad \text{pentru } j \leq m$$

## ► Ordinea de parcurgere a grafului de dependențe (ordinea de calcul)

$$i = 2, \dots, m; \quad j = i+1, \dots, n$$

# Descompunerea unui dreptunghi în pătrate

inițializări

```
for (i=2; i<=m; i++)
```

```
    for (j=i+1; j<=n; j++)
```

calculul lui  $a[i][j]$  conform relației de recurență

```
        if (j<=m)
```

```
             $a[j][i] = a[i][j]$ 
```

# Metoda Backtracking



# Metoda Backtracking

- ▶ Complexitatea în timp a algoritmilor joacă un rol esențial.
- ▶ **Un algoritm este considerat "acceptabil" numai dacă timpul său de executare este polinomial**

# Cadru

- ▶  $X = X_1 \times \dots \times X_n =$  **spațiul soluțiilor posibile (!vectori)**
- ▶  $\varphi: X \rightarrow \{0, 1\}$  este o **proprietate** definită pe  $X$
- ▶ **Căutăm un vector  $x \in X$  cu proprietatea  $\varphi(x)$** 
  - condiții interne pentru  $x$



# Cadru

- ▶ Generarea tuturor elementelor produsului cartezian  $X$  nu este acceptabilă.

Metoda backtracking încearcă micșorarea timpului de calcul.

# Metoda Backtracking

- ▶ Vectorul  $x$  este **construit progresiv**, începând cu prima componentă.
- ▶ Se avansează cu o valoare  $x_k$  dacă este satisfăcută **condiția de continuare**  $\varphi_k(x_1, \dots, x_k)$ .
- ▶ Condițiile de continuare rezultă de obicei din  $\varphi$ ; ele sunt strict necesare, **ideal fiind să fie și suficiente**.

# Metoda Backtracking

- ▶ Cazuri posibile la alegerea lui  $x_k$ :
  - ❑ Atribuire și avansează
  - ❑ Încercare eșuată
  - ❑ Revenire
  - ❑ Revenire după determinarea unei soluții

►  $C_k =$  mulțimea valorilor consumate din  $X_k$

$C_i \leftarrow \emptyset, \quad \forall i;$

$k \leftarrow 1;$

►  $C_k =$  mulțimea valorilor consumate din  $X_k$

$C_i \leftarrow \emptyset, \forall i;$

$k \leftarrow 1;$

while  $k > 0$

    if  **$k = n + 1$**

        retsol(x);  $k \leftarrow k - 1;$  {**revenire după o soluție**}

►  $C_k =$  mulțimea valorilor consumate din  $X_k$

$C_i \leftarrow \emptyset, \forall i;$

$k \leftarrow 1;$

while  $k > 0$

if  **$k = n + 1$**

$\text{retsol}(x); k \leftarrow k - 1; \{\text{revenire după o soluție}\}$

else

    if  $C_k \neq X_k$

**alege**  $v \in X_k \setminus C_k; C_k \leftarrow C_k \cup \{v\};$

►  $C_k =$  mulțimea valorilor consumate din  $X_k$

$C_i \leftarrow \emptyset, \forall i;$

$k \leftarrow 1;$

while  $k > 0$

if  **$k = n + 1$**

$\text{retsol}(x); k \leftarrow k - 1; \{ \text{revenire după o soluție} \}$

else

    if  **$C_k \neq X_k$**

**alege**  $v \in X_k \setminus C_k; C_k \leftarrow C_k \cup \{v\};$

        if  $\varphi_k(x_1, \dots, x_{k-1}, v)$

$x_k \leftarrow v; k \leftarrow k + 1; \{ \text{atribuie și avansează} \}$

►  $C_k =$  mulțimea valorilor consumate din  $X_k$

$C_i \leftarrow \emptyset, \forall i;$

$k \leftarrow 1;$

while  $k > 0$

if  **$k = n + 1$**

$\text{retsol}(x); k \leftarrow k - 1; \{ \text{revenire după o soluție} \}$

else

    if  **$C_k \neq X_k$**

        alege  $v \in X_k \setminus C_k; C_k \leftarrow C_k \cup \{v\};$

        if  $\varphi_k(x_1, \dots, x_{k-1}, v)$

$x_k \leftarrow v; k \leftarrow k + 1; \{ \text{atribuie și avansează} \}$

        else  $\{ \text{încercare eșuată} \}$



►  $C_k =$  mulțimea valorilor consumate din  $X_k$

$C_i \leftarrow \emptyset, \forall i;$

$k \leftarrow 1;$

while  $k > 0$

if  **$k = n + 1$**

$\text{retsol}(x); k \leftarrow k - 1; \{ \text{revenire după o soluție} \}$

else

    if  **$C_k \neq X_k$**

        alege  $v \in X_k \setminus C_k; C_k \leftarrow C_k \cup \{v\};$

        if  $\varphi_k(x_1, \dots, x_{k-1}, v)$

$x_k \leftarrow v; k \leftarrow k + 1; \{ \text{atribuie și avansează} \}$

        else  $\{ \text{încercare eșuată} \}$

    else  **$C_k \leftarrow \emptyset; k \leftarrow k - 1;$**   $\{ \text{revenire} \}$

► Dacă  $X_i = \{p_i, p_i+1, \dots, u_i\}$  algoritmul devine:

$x_i \leftarrow p_i - 1, \quad \forall i = 1, \dots, n$

$k \leftarrow 1;$

► Dacă  $X_i = \{p_i, p_i+1, \dots, u_i\}$  algoritmul devine:

$x_i \leftarrow p_i - 1, \quad \forall i=1, \dots, n$

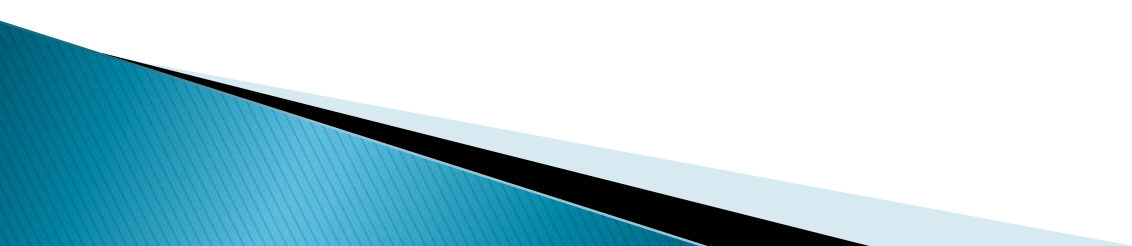
$k \leftarrow 1;$

while  $k > 0$

    if  $k = n+1$

        retsol(x);  $k \leftarrow k-1$ ; {**revenire după o sol.**}

    else



► Dacă  $X_i = \{p_i, p_i+1, \dots, u_i\}$  algoritmul devine:

$x_i \leftarrow p_i - 1, \quad \forall i=1, \dots, n$

$k \leftarrow 1;$

while  $k > 0$

    if  $k = n+1$

        retsol(x);  $k \leftarrow k-1$ ; { **revenire după o sol.** }

    else

        if  $\mathbf{x_k} < \mathbf{u_k}$

$x_k \leftarrow x_k + 1;$

► Dacă  $X_i = \{p_i, p_i+1, \dots, u_i\}$  algoritmul devine:

$x_i \leftarrow p_i - 1, \quad \forall i=1, \dots, n$

$k \leftarrow 1;$

while  $k > 0$

    if  $k = n+1$

        retsol(x);  $k \leftarrow k-1$ ; { **revenire după o sol.** }

    else

        if  $x_k < u_k$

$x_k \leftarrow x_k + 1;$

        if  $\varphi_k(x_1, \dots, x_k)$

$k \leftarrow k+1;$       { **atribuie și avansează** }

        else      { **încercare eșuată** }

► Dacă  $X_i = \{p_i, p_i+1, \dots, u_i\}$  algoritmul devine:

$x_i \leftarrow p_i - 1, \quad \forall i=1, \dots, n$

$k \leftarrow 1;$

while  $k > 0$

    if  $k = n+1$

        retsol(x);  $k \leftarrow k-1$ ; { **revenire după o sol.** }

    else

        if  $x_k < u_k$

$x_k \leftarrow x_k + 1;$

        if  $\varphi_k(x_1, \dots, x_k)$

$k \leftarrow k+1$ ;      { **atribuie și avansează** }

        else                      { **încercare eșuată** }

    else  $x_k \leftarrow p_k - 1$ ;  $k \leftarrow k-1$ ;      { **revenire** }

# Varianta recursivă

- ▶  $X_i = \{p_i, p_i + 1, \dots, u_i\}$
- ▶ Apelul inițial este: **back(1)**

```
procedure back(k)
```

```
  if k=n+1
```

```
    retsol
```

```
  else
```

```
    for (i=pk; i<=uk; i++) valori posibile
```

```
       $x_k \leftarrow i$ ;
```

```
      if  $\varphi_k(x_1, \dots, x_k)$ 
```

```
        back(k+1);
```

```
        revenire din recursivitate
```

```
end.
```



# Metoda Backtracking

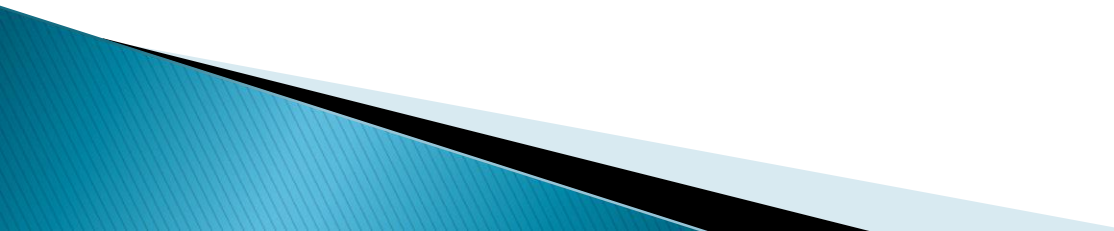
- ▶ Backtracking = parcurgerea limitată în adâncime a unui arbore



# Exemple

- ▶ Problema celor  $n$  dame
- ▶ Colorarea hărților
- ▶ Șiruri corecte de paranteze
- ▶ Permutări, combinări, aranjamente
- ▶ Problema ciclului hamiltonian

Pentru a testa condițiile de continuare  $\varphi_k(x_1, \dots, x_k)$   
vom folosi funcția `cont(k)`



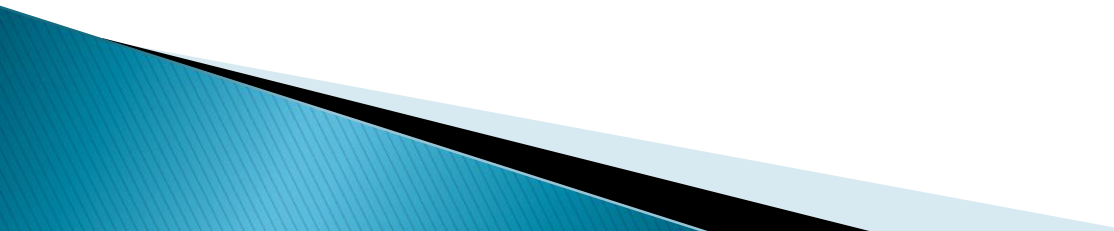
# Problema celor $n$ dame



► Se consideră un caroiaj  $n \times n$ .

Prin analogie cu o tablă de șah ( $n=8$ ), se dorește plasarea a  $n$  dame pe pătrățelele caroiajului, astfel încât să nu existe două dame una în bătaia celeilalte (adică să nu existe două dame pe aceeași linie, coloană sau diagonală).

# Problema celor $n$ dame

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

# Problema celor n dame

## ► Reprezentarea soluției

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

$\mathbf{x}_k$  = coloana pe care este plasată dama de pe linia  $k$

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

## ► Condiții interne (finale)

pentru orice  $i \neq j$ :  $\mathbf{x}_i \neq \mathbf{x}_j$  și  $|\mathbf{x}_i - \mathbf{x}_j| \neq |j - i|$

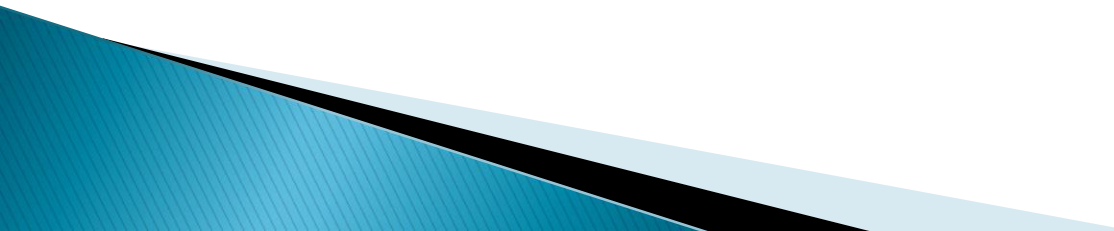
## ► Condiții de continuare

pentru orice  $i < k$ :  $\mathbf{x}_i \neq \mathbf{x}_k$  și  $|\mathbf{x}_i - \mathbf{x}_k| \neq k - i$

# Problema celor n dame

```
boolean cont(int k){
    for(int i=1;i<k;i++)
        if((x[i]==x[k]) || (Math.abs(x[k]-x[i])==k-i))
            return false;
    return true;
}

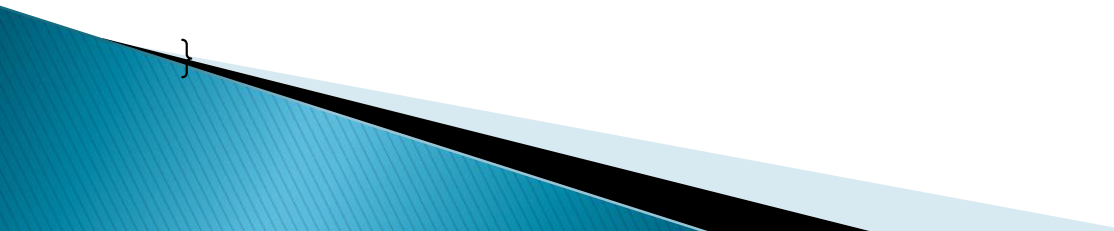
void retsol(int[] x){
    for(int i=1;i<=n;i++)
        System.out.print("(" + i + ", " + x[i] + ") ");
    System.out.println();
}
```



# Problema celor n dame

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

```
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]<n) {  
                x[k]++; // atribuie  
                if (cont(k)) k++; // si avanseaza  
            }  
            else { x[k]=0; k--; } // revenire  
        }  
    }  
}
```



# Colorarea hărților

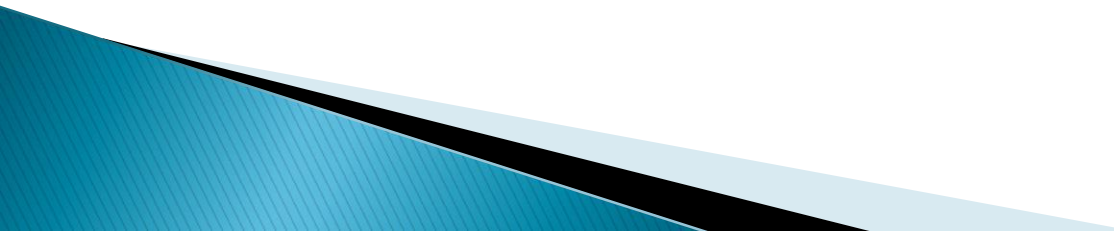


► Se consideră o hartă cu  $n$  țări.

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)

$\mathbf{x}_i \neq \mathbf{x}_j$  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  $k$

```
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;  
            if (cont(k))  
                backrec(k+1);  
        }  
}
```

```

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--; }
        else{
            if(x[k]<4) {
                x[k]++;
                if (cont(k))
                    k++;
            }
            else{ x[k]=0; k--; }
        }
    }
}

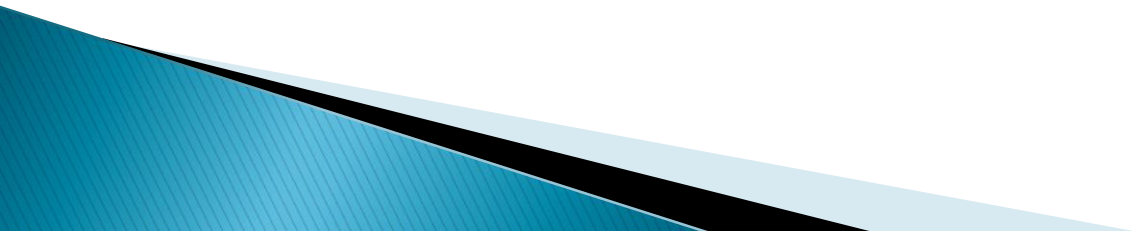
```

# Șiruri corecte de paranteze



- ▶ Să se genereze toate șirurile de  $n$  paranteze ce se închid corect ( $n$  par)

# Șiruri corecte de paranteze

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

# Șiruri corecte de paranteze

## ► Reprezentarea soluției

$$\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}, \text{ unde} \\ \mathbf{x}_k \in \{0, 1\}$$

$$\text{Notăm } \text{dif} = \text{nr}_( - \text{nr}_)$$

## ► Condiții interne (finale)

## ► Condiții de continuare

# Șiruri corecte de paranteze

## ► Reprezentarea soluției

$\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$ , unde  
 $\mathbf{x}_k \in \{0, 1\}$

Notăm  $\text{dif} = \text{nr}_( - \text{nr}_)$

## ► Condiții interne (finale)

$\text{dif} = 0$

$\text{dif} \geq 0$  pentru orice secvență  $\{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k\}$

## ► Condiții de continuare

$\text{dif} \geq 0 \rightarrow$  doar necesar



# Șiruri corecte de paranteze

## ► Reprezentarea soluției

$\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$ , unde  
 $\mathbf{x}_k \in \{ ' ( ' , ' ) ' \}$

Notăm  $\text{dif} = \text{nr}_(' - \text{nr}_)$

## ► Condiții interne (finale)

$\text{dif}=0$

$\text{dif} \geq 0$  pentru orice secvență  $\{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k\}$

## ► Condiții de continuare

$\text{dif} \geq 0 \quad \rightarrow$  doar necesar

$\text{dif} \leq n-k \quad \rightarrow$  și suficient

```

void back() {
    dif=0;
    back(1);
}
void back(int k) {
    if(k==n+1)
        retsol(x);
    else{
        x[k]='(';
        dif++;
        if (dif <= n-k)
            back(k+1);
        dif--;

        x[k]=')';
        dif--;
        if (dif >= 0)
            back(k+1);
        dif++;
    }
}

```

# Metoda Backtracking

- ▶ **Variantele** cele mai uzuale întâlnite în aplicarea metodei backtracking sunt următoarele:
  - soluția poate avea un număr variabil de componente *și/sau*
  - dintre ele alegem una care optimizează o funcție dată
- ▶ **Exemplu:** Fie  $a=(a_1,\dots,a_n)\in\mathbf{Z}^n$ . Să se determine un subșir crescător de lungime maximă.

Completăm cu  $-\infty$  și  $+\infty$  :  $a_0 \leftarrow -\infty$ ;  $n \leftarrow n+1$ ;  $a_n \leftarrow +\infty$ .

## ► Reprezentarea soluției

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

## ► Condiții interne (finale)

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

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

## ► Condiții de continuare

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

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

```
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;  
    s=0;  
    backrec(1);  
}
```

```

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

```

```

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

```

## Backtracking în plan

- ▶ Se consideră un caroiăj (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.



## Backtracking în plan

- ▶ **Mișcările posibile** sunt date printr-o matrice  $dep1$  cu două linii și  $ndep1$  coloane. De exemplu, dacă deplasările permise sunt cele către pozițiile vecine situate la Est, Nord, Vest și Sud, matricea are forma:

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

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

## Backtracking în plan

- ▶ Dacă poziția este liberă și putem continua, setăm  $a_{ij} = -1$  (a fost atinsă), continuăm și apoi repunem  $a_{ij} \leftarrow 0$  (întoarcere din recursivitate).
- ▶ Pentru refacerea drumurilor reținem un vector cu pozițiile parcurse sau pentru fiecare poziție atinsă memorăm legătura la precedenta.

```

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)
                print(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( $i_0, j_0$ )