Examen

- **242, 243 – la curs** Miercuri 15.01.2014
- **241, 244 – la seminar**
  - **244** – Vineri 17.01.2014 , 10-12
  - **241** – Vineri 17.01.2014 , 12-14

- **242, 243 – la curs** Miercuri 15.01.2014
- **241, 244 – la seminar**
  - **244** – Vineri 17.01.2014 , 10-12
  - **241** – Vineri 17.01.2014 , 12-14

Modificări laborator vineri 17.01.2014

- **244** – Miercuri 14-16
- **243** – Luni 14-16

- Java
  - elemente fundamentale
  - clase, tablouri, citirea de la tastatură
  - colecții, tipuri generice

- Tehnici de programare
  - Divide et Impera (complexitate)
  - Greedy (corectitudine)
  - Programare dinamică
  - Backtracking

- Algoritmi genetici

Algoritmi Probabiliști

Algoritmi Probabiliști

- În timpul rezolvării unei probleme, putem ajunge la un moment dat în situaţia de a avea **de ales** între mai multe variante de continuare.

Algoritmi probabiliști

- Monte Carlo
- Las Vegas
- numerici

Algoritmi Monte Carlo

- Furnizează totdeauna un rezultat, care însă **nu este neapărat corect**
- **Probabilitatea ca rezultatul să fie corect creşte pe măsură ce timpul disponibil creşte**

Algoritmi Monte Carlo

Se consideră un vector cu *n* elemente distincte. Să se determine un element al vectorului care să fie mai mare sau egal cu media aritmetică a celor n numere din vector

Algoritmi Monte Carlo

Repetă fără a depăşi timpul disponibil:

- alegem aleator un element al vectorului
- păstrăm într-o variabilă v cel mai mare dintre elementele alese

Algoritmi Monte Carlo

- Dacă în timpul disponibil am analizat k elemente, probabilitatea ca toate să fie mai mici decât mediana este $1/2 * 1/2 * \ldots * 1/2 = 1/2^k$
- Probabilitatea ca valoarea să fie corectă este

$$1 - 1/2^k$$

- Pentru k=20, această probabilitate este mai mare decât 0,999999.

Algoritmi Las Vegas

- **Nu furnizează totdeauna un rezultat**, dar dacă furnizează un rezultat atunci acesta este **corect**

Algoritmi Las Vegas

- **Nu furnizează totdeauna un rezultat**, dar dacă furnizează un rezultat atunci acesta este **corect**

- **Probabilitatea ca algoritmul să se termine crește pe măsură ce timpul disponibil crește**

Algoritmi Las Vegas

Se dau n texte (n foarte mare) cu următoarele proprietăţi:

- **există un unic text t_0 care apare de cel puţin 10% ori;**
- **celelalte texte sunt distincte.**

Se cere determinarea textului t_0.

Algoritmi Las Vegas

```
repeat
   i ← random(1..n); j ← random(1..n);
   if i≠j and t_i=t_j
     write t_i; stop
until false
```

Algoritmi Las Vegas

Probabilitatea algoritmul să se termine:

Algoritmi Las Vegas

Probabilitatea algoritmul să se termine:

- probabilitatea ca $t_i = t_0$

Algoritmi Las Vegas

Probabilitatea algoritmul să se termine:

- probabilitatea ca $t_i = t_0 \longrightarrow 1/10$

Algoritmi Las Vegas

Probabilitatea algoritmul să se termine:

- probabilitatea ca $t_i = t_0 \longrightarrow 1/10$
- probabilitatea ca $t_j = t_0 \longrightarrow 1/10$

Algoritmi Las Vegas

Probabilitatea algoritmul să se termine:

- probabilitatea ca $t_i = t_0 \longrightarrow 1/10$
- probabilitatea ca $t_j = t_0 \longrightarrow 1/10$
- probabilitatea ca $t_i = t_j = t_0$ este

Algoritmi Las Vegas

Probabilitatea algoritmul să se termine:

- probabilitatea ca $t_i = t_0 \longrightarrow 1/10$
- probabilitatea ca $t_j = t_0 \longrightarrow 1/10$
- probabilitatea ca $t_i = t_j = t_0$ este

$$1/10 * 1/10 = 1/100$$

Algoritmi Las Vegas

Probabilitatea algoritmul să se termine:

- probabilitatea ca $t_i = t_0 \longrightarrow 1/10$
- probabilitatea ca $t_j = t_0 \longrightarrow 1/10$
- probabilitatea ca $t_i = t_j = t_0$ este

$$1/10 * 1/10 = 1/100$$

Teoretic sunt suficiente 100 de încercări, independent de valoarea lui n

Algoritmi Las Vegas

Problema celor n dame

Se consideră un caroiaj n×n. Prin analogie cu o tablă de şah (n=8), se doreşte plasarea a n dame pe pătrăţelele caroiajului, astfel încât să nu existe două dame una în bătaia celeilalte

http://backtracking.xhost.ro/prcd.html

Algoritmi Las Vegas - Problema damelor

- **Reprezentarea soluţiei**

  $\mathbf{x} = \{\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_n}\}$, unde

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

  $\mathbf{x_k} \in \{1, 2, \ldots, n\}$

Algoritmi Las Vegas – Problema damelor

Algoritm probabilist

- plasăm o damă pe prima linie;

Algoritmi Las Vegas – Problema damelor

Algoritm probabilist

- plasăm o damă pe prima linie;
- presupunând că am plasat câte o damă pe liniile 1,...,k-1, facem o listă a poziţiilor posibile pentru dama de pe linia k

Algoritmi Las Vegas – Problema damelor

Algoritm probabilist

- plasăm o damă pe prima linie;
- presupunând că am plasat câte o damă pe liniile 1,...,k-1, facem o listă a poziţiilor posibile pentru dama de pe linia k
  - Dacă lista este nevidă, alegem **aleator** o poziție din listă

Algoritmi Las Vegas – Problema damelor

Algoritm probabilist

- plasăm o damă pe prima linie;
- presupunând că am plasat câte o damă pe liniile 1,...,k−1, facem o listă a poziţiilor posibile pentru dama de pe linia k
  - Dacă lista este nevidă, alegem **aleator** o poziție din listă
  - Altfel reluăm **întreg** algoritmul (**!** nu ne întoarcem la linia precedentă)

Algoritmi Las Vegas – Problema damelor

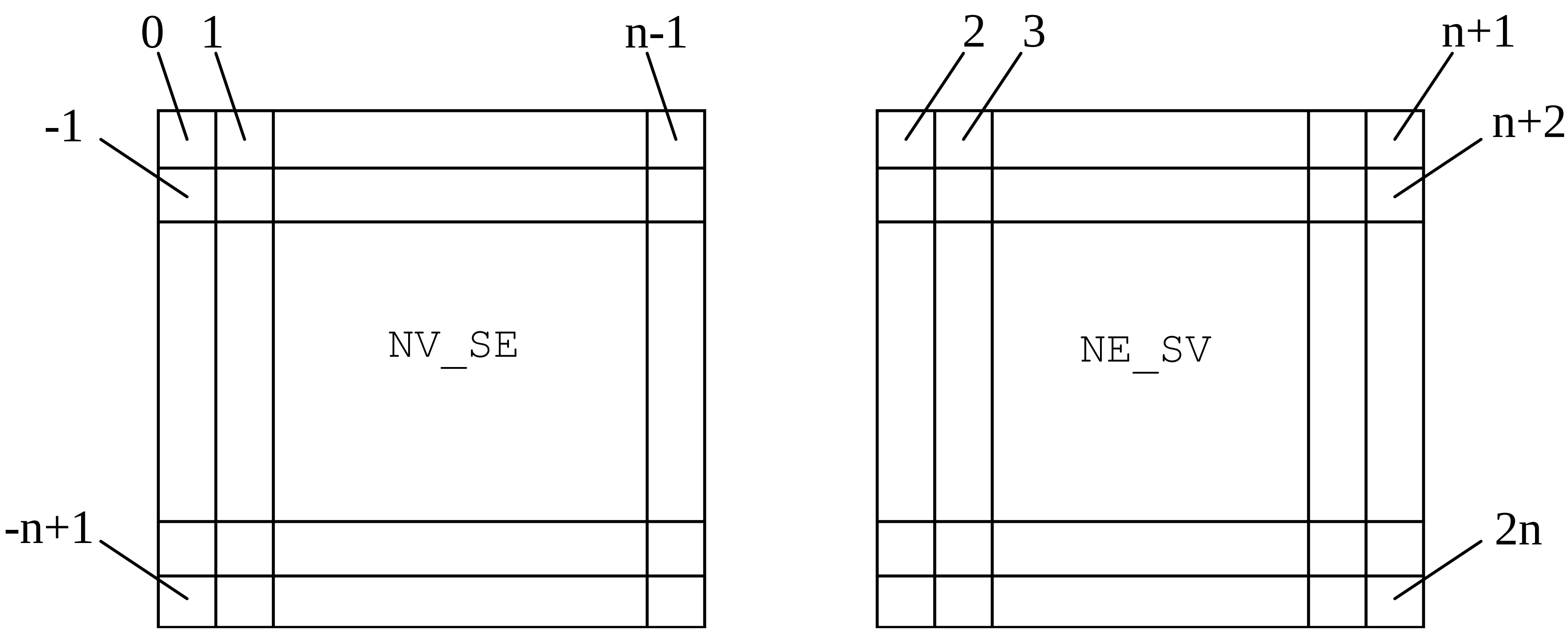
Algoritm probabilist

- plasăm o damă pe prima linie;
- presupunând că am plasat câte o damă pe liniile 1,...,k–1, facem o listă a poziţiilor posibile pentru dama de pe linia k
  - Dacă lista este nevidă, alegem **aleator** o poziție din listă
  - Altfel reluăm **întreg** algoritmul (**!** nu ne întoarcem la linia precedentă)
- Dacă am plasat o damă pe linia n, atunci am determinat o soluţie şi oprim programul

Algoritmi Las Vegas – Problema damelor

Pentru a ține o evidenţă a coloanelor şi diagonalelor ocupate – vectorii:

- **NV_SE[−n+1..n−1]** (j – i = constant)
- **NE_SV[2..2n]** (j + i = constant)
- **C[1..n]**



Algoritmi Las Vegas – Problema damelor

```
repeat
   repeat
      • iniţializăm componentele celor 3 vectori C,
        NV_SE, NE_SV cu valoarea true
      • k ← 1
      • facem inventarul poziţiilor i∈{1,...,n} cu
             C[i] = NV_SE[i-k] = NE_SV[i+k] = true
      • plasăm aceste poziţii în primele na componente
        ale unui vector a
      •







   until na=0 ∨ k=n+1
until k=n+1
write(x)
```

Algoritmi Las Vegas – Problema damelor

```
repeat
   repeat
      • inițializăm componentele celor 3 vectori C,
        NV_SE, NE_SV cu valoarea true
      • k ← 1
      • facem inventarul pozițiilor i∈{1,...,n} cu
            C[i] = NV_SE[i-k] = NE_SV[i+k] = true
      • plasăm aceste poziții în primele na componente
        ale unui vector a
      • if na>0 then
            aleg aleator i∈{1,...,na};  i ← a_i
            x_k ← i ;
            NV_SE[i-k] ← false; NE_SV[i+k] ← false;
            C_i ← false;
            k ← k+1
   until na=0 ∨ k=n+1
until k=n+1
write(x)
```

Algoritmi numerici

Algoritmi numerici

- Urmăresc determinarea aproximativă a unei valori
- **Cu cât timpul alocat executării algoritmului este mai mare, precizia rezultatului se îmbunătăţeşte**

Algoritmi numerici

- **Aproximarea** lui π
- **Aproximarea** $\int_a^b f(x)\,dx$

$$f:[a,b]\to[c,d]$$

Aproximarea lui π

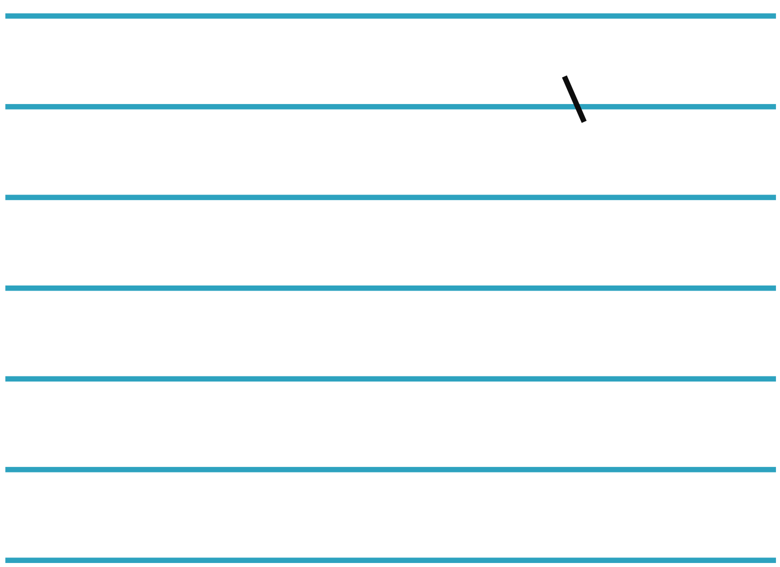
1. Acul lui Buffon

Se consideră o mulţime de linii paralele astfel încât oricare două linii vecine sunt la distanţă de o unitate.

Aproximarea lui π

1. Acul lui Buffon

Se consideră o mulţime de linii paralele astfel încât oricare două linii vecine sunt la distanţă de o unitate. Un ac de lungime o jumătate de unitate este aruncat aleator şi se numără de câte ori a intersectat vreo linie.



Aproximarea lui π

1. Acul lui Buffon

Se consideră o mulţime de linii paralele astfel încât oricare două linii vecine sunt la distanţă de o unitate. Un ac de lungime o jumătate de unitate este aruncat aleator şi se numără de câte ori a intersectat vreo linie.

- **Probabilitatea ca acul să intersecteze o linie este $1/\pi$**

Aproximarea lui π

1. Acul lui Buffon

Se consideră o mulţime de linii paralele astfel încât oricare două linii vecine sunt la distanţă de o unitate. Un ac de lungime o jumătate de unitate este aruncat aleator şi se numără de câte ori a intersectat vreo linie.

- **Probabilitatea ca acul să intersecteze o linie este $1/\pi$**
- După un număr "suficient de mare" de încercări, raportul între:

  - **numărul total de încercări**

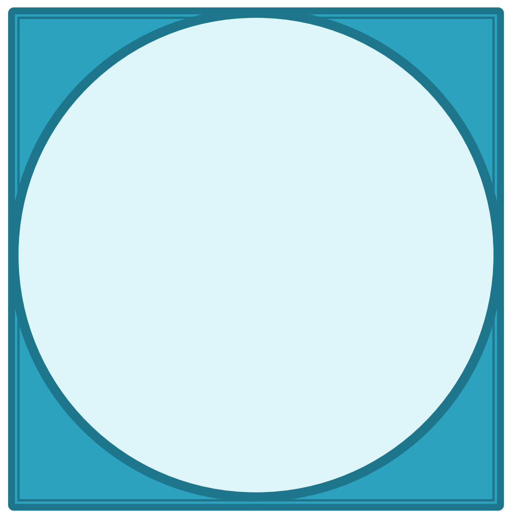
  - numărul cazurilor în care acul a intersectat vreo linie

va fi "suficient de aproape" de π.

Aproximarea lui π

2. Se aruncă repetat cu o săgeată într-un panou pătrat, cu ținta un cerc înscris în pătrat.

Se presupune că săgeata nimereşte totdeauna panoul.



Aproximarea lui π

2. Se aruncă repetat cu o săgeată într-un panou pătrat, cu ținta un cerc înscris în pătrat.

Se presupune că săgeata nimereşte totdeauna panoul.

Atunci raportul dintre:

- numărul cazurilor în care săgeata nimereşte în cercul înscris în pătrat

- numărul total de încercări

tinde la

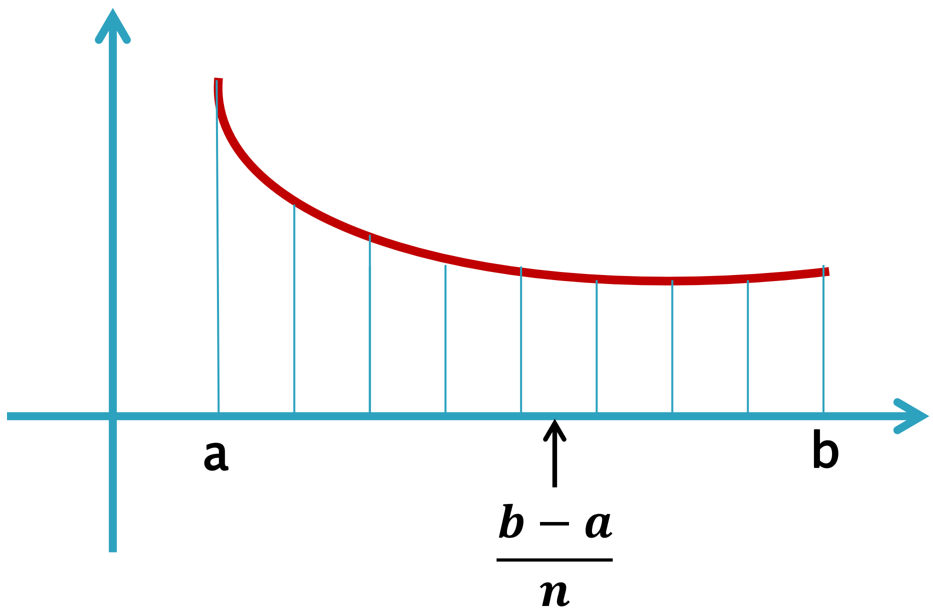
$$\frac{arie\ cerc}{arie\ patrat} = \frac{\pi r^2}{(2r)^2} = \frac{\pi}{4}$$

Aproximarea integralei

$$\int_a^b f(x)\,dx, \quad f:[a,b]\to[c,d]$$

Aproximarea integralei

$$\int_a^b f(x)\,dx, \quad f:[a,b]\to[c,d]$$



Aproximarea integralei

$$\int_a^b f(x)\,dx, \quad f:[a,b]\to[c,d]$$

```
s ← 0
for i=1,n
   x ← random([a,b]);
   s ← s+f(x)

s ← s·(b-a)/n
write(s)
```

Metoda Backtracking

Metoda Backtracking

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

Cadru

- $X=X_1 \times \ldots \times X_n$ = **spaţiul soluţiilor posibile (!vectori)**
- $\varphi: X \to \{0,1\}$ este o **proprietate** definită pe X
- **Căutăm un vector $x \in X$ cu proprietatea $\varphi(x)$**
  - condiţii interne pentru x

Cadru

- Generarea tuturor elementelor produsului cartezian X nu este acceptabilă.

Cadru

- Generarea tuturor elementelor produsului cartezian X nu este acceptabilă.

  Metoda backtracking încearcă micşorarea timpului de calcul – prin **evitarea generării unor soluţii care nu satisfac condiţiile interne**

Metoda Backtracking

- Vectorul x este **construit progresiv**, începând cu prima componentă.

Metoda Backtracking

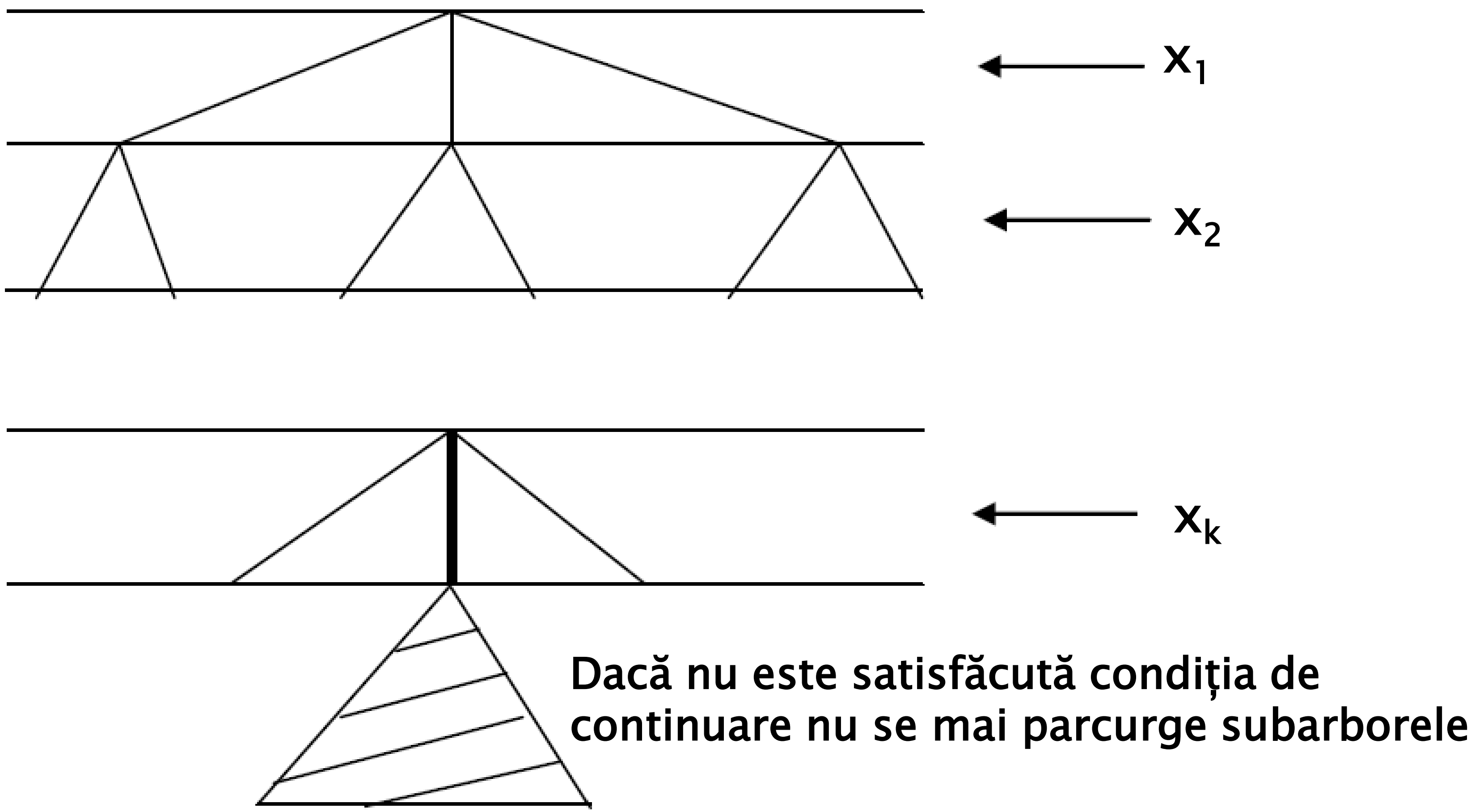
- Vectorul x este **construit progresiv**, începând cu prima componentă.
- Se avansează cu o valoare x_k dacă este satisfăcută o **condiţie de continuare** $\varphi_k(x_1,\dots,x_k)$

Metoda Backtracking

- Vectorul x este **construit progresiv**, începând cu prima componentă.
- Se avansează cu o valoare x_k dacă este satisfăcută o **condiţie de continuare** $\varphi_k(x_1,\ldots,x_k)$
- **Condiţiile de continuare rezultă de obicei din** φ Ele sunt **strict necesare**, **ideal fiind să fie şi suficiente.**

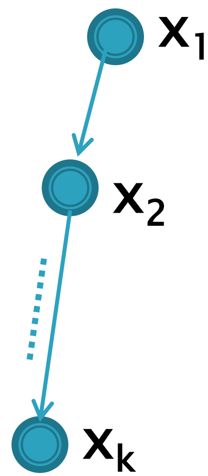
Metoda Backtracking

- Backtracking = parcurgerea <u>limitată</u> în adâncime a unui arbore



Metoda Backtracking

- Cazuri posibile la alegerea lui x_k:

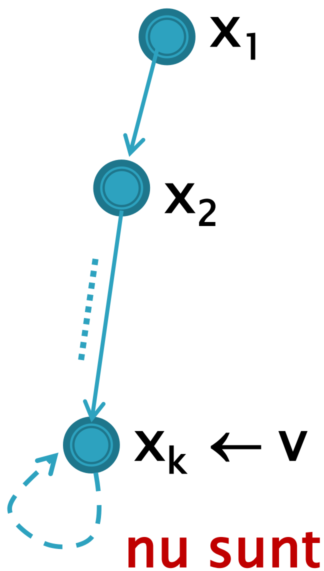


Metoda Backtracking

- Atribuie şi avansează

Metoda Backtracking

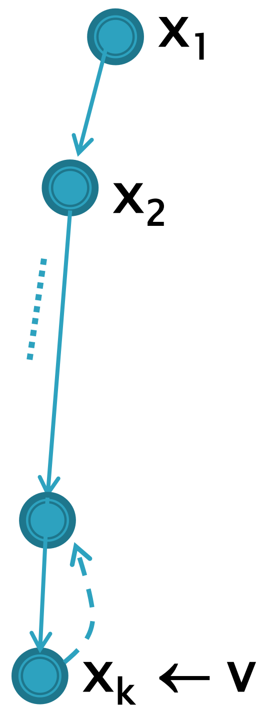
- Încercare eşuată



nu sunt verificate condiţiile de continuare

Metoda Backtracking

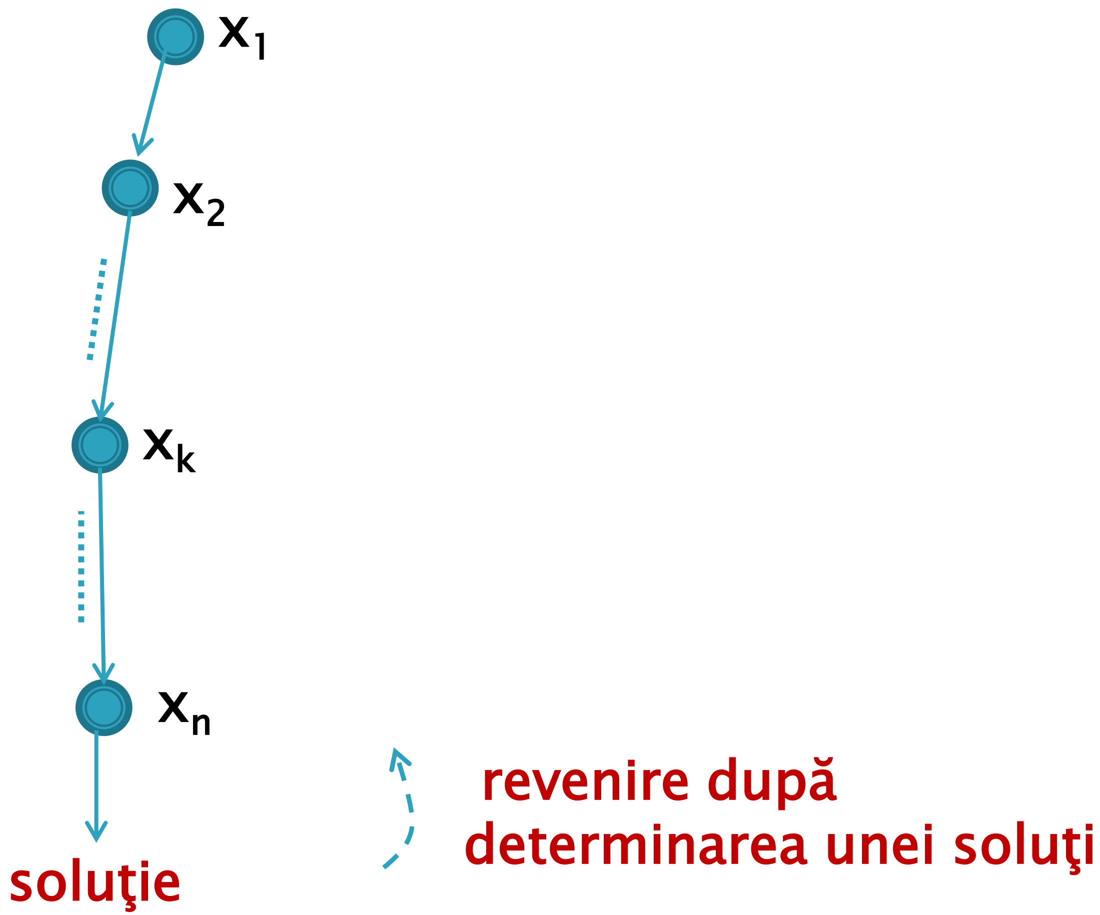
- Revenire



nu mai există valori pentru x_k neconsiderate

Metoda Backtracking

- Revenire după determinarea unei soluţii



Varianta nerecursiva - pseudocod

- C_k = mulţimea valorilor consumate din X_k

```
Cᵢ←∅, ∀i;
k←1;
```

- C_k = mulţimea valorilor consumate din X_k

```
Ci←∅, ∀i;
k←1;
while k>0
  if k=n+1
     retsol(x); k←k-1;{revenire după o soluţie}
```

- C_k = mulţimea valorilor consumate din X_k

```
Cᵢ←∅, ∀i;
k←1;
while k>0
  if k=n+1
     retsol(x); k←k-1;{revenire după o soluţie}
  else
     if Cₖ ≠ Xₖ
        alege v∈Xₖ\Cₖ; Cₖ←Cₖ∪{v};
```

- C_k = mulţimea valorilor consumate din X_k

```
Ci←∅, ∀i;
k←1;
while k>0
  if k=n+1
     retsol(x); k←k-1;{revenire după o soluţie}
  else
     if Ck ≠ Xk
        alege v∈Xk\Ck; Ck←Ck∪{v};
        if φk(x1,…,xk-1,v)
           xk←v; k←k+1; { atribuie şi avansează }
```

- C_k = mulţimea valorilor consumate din X_k

```
Ci←∅, ∀i;
k←1;
while k>0
  if k=n+1
    retsol(x); k←k-1;{revenire după o soluţie}
  else
     if Ck ≠ Xk
        alege v∈Xk\Ck; Ck←Ck∪{v};
        if φk(x1,…,xk-1,v)
           xk←v; k←k+1; { atribuie şi avansează }
        else                       { încercare eşuată }
```

- C_k = mulţimea valorilor consumate din X_k

```
Cᵢ←∅, ∀i;
k←1;
while k>0
  if k=n+1
    retsol(x); k←k-1;{revenire după o soluţie}
  else
     if Cₖ ≠ Xₖ
       alege v∈Xₖ\Cₖ; Cₖ←Cₖ∪{v};
       if φₖ(x₁,…,xₖ₋₁,v)
          xₖ←v; k←k+1; { atribuie şi avansează }
       else                    { încercare eşuată }
    else Cₖ←∅; k←k-1;           { revenire }
```

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

$x_i \leftarrow \mathbf{p_i - 1}$, $\forall i=1,\dots,n$

$k \leftarrow 1$;

- Dacă $\mathbf{X_i} = \{\mathbf{p_i}, \mathbf{p_i+1}, \ldots, \mathbf{u_i}\}$ algoritmul devine:

```
xi←pi-1, ∀i=1,...,n
k←1;
while k>0
   if k=n+1
        retsol(x); k←k-1;{revenire după o sol.}
   else
```

- Dacă $\mathbf{X_i}$= {$\mathbf{p_i}$, $\mathbf{p_i}$+1,...,$\mathbf{u_i}$} algoritmul devine:

```
xi←pi-1, ∀i=1,...,n
k←1;
while k>0
   if k=n+1
        retsol(x); k←k-1;{revenire după o sol.}
   else
       if xk<uk
           xk←xk+1;
```

- Dacă $\mathbf{X_i = \{p_i, p_i+1, \ldots, u_i\}}$ algoritmul devine:

```
xᵢ←pᵢ-1, ∀i=1,...,n
k←1;
while k>0
   if k=n+1
       retsol(x); k←k-1;{revenire după o sol.}
   else
       if xₖ<uₖ
           xₖ←xₖ+1;
           if φₖ(x₁,…,xₖ)
               k←k+1;    { atribuie şi avansează }
           else           { încercare eşuată }
```

- Dacă $\mathbf{X_i} = \{\mathbf{p_i}, \mathbf{p_i}+1,\ldots,\mathbf{u_i}\}$ algoritmul devine:

```
xᵢ←pᵢ-1, ∀i=1,...,n
k←1;
while k>0
   if k=n+1
       retsol(x); k←k-1;{revenire după o sol.}
   else
       if xₖ<uₖ
           xₖ←xₖ+1;
           if φₖ(x₁,…,xₖ)
               k←k+1;    { atribuie şi avansează }
           else          { încercare eşuată }
       else xₖ←pₖ-1; k←k-1;    { revenire }
```

Varianta recursivă

- $X_i = \{p_i, p_i+1, \ldots, u_i\}$
- Apelul iniţial este: **`back(1)`**

```
procedure back(k)
  if k=n+1
        retsol(x)
  else





end.
```

- $X_i = \{p_i, p_i+1, \ldots, u_i\}$
- Apelul iniţial este: **back(1)**

```
procedure back(k)
  if k=n+1
      retsol(x)
  else
      for (i=pk; i<=uk; i++) //valori posibile
          xk←i;




end.
```

- $X_i = \{p_i, p_i+1, \ldots, u_i\}$
- Apelul iniţial este: **`back(1)`**

```
procedure back(k)
  if k=n+1
       retsol(x)
  else
       for (i=p_k; i<=u_k; i++) {valori posibile}
          x_k←i;
          if φ_k(x_1,…,x_k)
              back(k+1);
              {revenire din recursivitate}
end.
```

Exemple

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

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

Permutări

- Reprezentarea soluţiei

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

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

- Condiţii interne (finale)

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

- Condiţii de continuare (!!pentru x_k)

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

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 ţări.

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

Colorarea hărţilor

- Reprezentarea soluţiei

- Condiţii interne (finale)

- Condiţii de continuare (!!pentru x_k)

Colorarea hărţilor

- **Reprezentarea soluţiei**

  $\mathbf{x} = \{\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_n}\}$, unde

  $\mathbf{x_k}$ = culoarea cu care este colorată ţara $\mathbf{k}$

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

- **Condiţii interne (finale)**

- **Condiţii de continuare (!!pentru x_k)**

Colorarea hărţilor

- **Reprezentarea soluţiei**

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

  $\mathbf{x}_k$ = culoarea cu care este colorată ţara **k**

  $\mathbf{x}_k \in \{1,2,3,4\} \quad (\mathbf{p}_k = 1,\ \mathbf{u}_k = 4)$.

- **Condiţii interne (finale)**

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

- **Condiţii de continuare (!!pentru x_k)**

Colorarea hărţilor

- **Reprezentarea soluţiei**

  $\mathbf{x = \{x_1, x_2, \ldots, x_n\}}$, unde

  $\mathbf{x_k}$ = culoarea cu care este colorată ţara **k**

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

- **Condiţii interne (finale)**

  $\mathbf{x_i \neq x_j}$ pentru orice două ţări **vecine** i şi j.

- **Condiţii de continuare (!!pentru x_k)**

  $\mathbf{x_i \neq x_k}$ pentru orice ţară $\mathbf{i \in \{1,2,\ldots,k-1\}}$ vecină cu ţara 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++){




        }
}
```

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
                }
        }
}
```

Problema celor n dame

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

Prin analogie cu o tablă de şah ($n=8$), se doreşte plasarea a n dame pe pătrăţelele caroiajului, astfel încât să nu existe două dame una în bătaia celeilalte

Problema celor n dame

- **Reprezentarea soluţiei**

- **Condiţii interne (finale)**

- **Condiţii de continuare**

Problema celor n dame

- **Reprezentarea soluţiei**

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

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

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

- **Condiţii interne (finale)**

- **Condiţii de continuare**

Problema celor n dame

- Reprezentarea soluţiei

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

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

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

- Condiţii interne (finale)

  pentru orice $i \neq j$: $x_i \neq x_j$ și $|x_i - x_j| \neq |j - i|$

- Condiţii de continuare

Problema celor n dame

- **Reprezentarea soluţiei**

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

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

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

- **Condiţii interne (finale)**

  pentru orice $i \neq j$: $x_i \neq x_j$ și $|x_i - x_j| \neq |j - i|$

- **Condiţii de continuare**

  pentru orice $i < k$: $x_i \neq x_k$ și $|x_i - x_k| \neq k - i$

Implementare – varianta recursivă

```
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;
}
```

Implementare - varianta recursivă

```
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();
}
```

Implementare – varianta recursivă

```
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);
          }
}
```

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

Șiruri corecte de paranteze

- Să se genereze toate şirurile de n paranteze ce se închid corect (n par)

Şiruri corecte de paranteze

- Reprezentarea soluţiei

- Condiţii interne (finale)

- Condiţii de continuare

Șiruri corecte de paranteze

- Reprezentarea soluţiei

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

  $\mathbf{x}_k \in \{\text{'('}, \text{')'}\}$

- Condiţii interne (finale)

- Condiţii de continuare

Șiruri corecte de paranteze

- **Reprezentarea soluţiei**

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

  $\mathbf{x}_k \in \{$'(', ')'$\}$

- **Condiţii interne (finale)**

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

  $\mathbf{dif} = 0$

  $\mathbf{dif} \geq 0$ pentru orice secvenţă $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_k\}$

- **Condiţii de continuare**

Șiruri corecte de paranteze

- **Reprezentarea soluţiei**

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

  $\mathbf{x}_k \in \{\texttt{'('}, \texttt{')'}\}$

- **Condiţii interne (finale)**

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

  $\mathbf{dif} = 0$

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

- **Condiţii de continuare**

  $\mathbf{dif} \geq 0$ -> doar necesar

Șiruri corecte de paranteze

- **Reprezentarea soluţiei**

  $x = \{x_1, x_2, \dots, x_n\}$, unde

  $x_k \in \{'(', ')'\}$

- **Condiţii interne (finale)**

  Notăm $dif = nr_(- nr_)$

  $dif = 0$

  $dif \geq 0$ pentru orice secvenţă $\{x_1, x_2, \dots, x_k\}$

- **Condiţii de continuare**

  $dif \geq 0$ -> doar necesar

  $dif \leq n-k$ -> şi suficient

```
void back(){
     dif=0;
     back(1);
}
void back(int k){
     if(k==n+1)
          retsol(x);
     else{











     }
}
```

```
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--;





     }
}
```

```
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:
  - soluţia poate avea un număr variabil de componente *şi/sau*
  - dintre ele alegem una care optimizează o funcţie dată

Metoda Backtracking

- **Exemplu**: Dat un număr natural n, să se genereze toate partițiile lui n ca sumă de numere pozitive

Partiție a lui n = $\{x_1, x_2, \ldots, x_k\}$ cu

$$x_1 + x_2 + \ldots + x_k = n$$

- Reprezentarea soluţiei

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

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

- Condiţii interne (finale)

- Condiţii de continuare

- Reprezentarea soluţiei

  $x = \{x_1, x_2, \ldots, x_k\}$, unde

  $x_i \in \{1, \ldots, n\}$

- Condiţii interne (finale)

  $x_1 + x_2 + \ldots + x_k = n$

  Pentru unicitate: $x_1 \le x_2 \le \ldots \le x_k$

- Condiţii de continuare

Reprezentarea soluţiei

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

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

Condiţii interne (finale)

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

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

Condiţii de continuare

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

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

Implementare – varianta recursivă

```java
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)////verif.cond.de cont









          // else return;
     }
}
```

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







        // else return;
    }
}
```

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

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 - verif. 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 - verif. 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 - verif. 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 - verif. 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
         }
      }
   }
}
```

Backtracking în plan

- Se consideră un caroiaj (matrice) A cu m linii şi n coloane. Poziţiile pot fi:
  - libere: $a_{ij}=0$;
  - ocupate: $a_{ij}=1$.

  Se mai dă o poziţie (i_0,j_0). Se caută **toate** drumurile care ies în afara matricei, trecând numai prin poziţii libere.

Backtracking în plan

- **Mişcările posibile** sunt date printr-o matrice `depl` cu două linii şi `ndepl` coloane. De exemplu, dacă deplasările permise sunt cele către poziţiile vecine situate la E, N, S:

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

Backtracking în plan

- **Mişcările posibile** sunt date printr-o matrice `depl` cu două linii şi `ndepl` coloane. De exemplu, dacă deplasările permise sunt cele către poziţiile vecine situate la E, N, S:

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

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

- **Reprezentarea soluţiei**

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

  $\mathbf{x}_i$ = a i-a celulă din drum

- **Condiţii interne (finale)**

- **Condiţii de continuare**

- **Reprezentarea soluţiei**

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

  $\mathbf{x}_i$ = a i-a celulă din drum

- **Condiţii interne (finale)**

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

- **Condiţii de continuare**

- **Reprezentarea soluţiei**

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

  $\mathbf{x}_i$ = a i-a celulă din drum

- **Condiţii interne (finale)**

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

- **Condiţii de continuare**

  $\mathbf{x}_k$ celulă liberă (cu 0) **prin care nu am mai trecut**

Backtracking în plan

- **dacă poziţia este liberă şi putem continua, setăm a_{ij}=-1** (a fost **atinsă**), continuăm
- repunem a_{ij}=0 la întoarcere (din recursivitate).

```
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
                    xk ← (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
               xk ← (ii, jj);
               a[i][j] = -1; //marcam
               back(ii, jj);
               a[i][j] = 0;  //demarcam
               k = k-1 ;     //scade
            }
   }
}
```

Apel:

```
x₁ ← (i0, j0);
k = 1;
back(i0, j0)
```