Problema monedelor

- Avem la dispoziție un număr nelimitat de monede de valori $\{v_1,v_2,\ldots,v_n\}$ şi 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} şi 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 plati o sumă $s \leq S$.

- **Soluție** **`nr[S]`**

Problema monedelor

- **Ştim direct**

  `nr[0] = 0`

- **Relaţie de recurenţă**

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

- **Ordinea de calcul**

  `s = 1,…,S` -> Complexitate O(nS)

- **Memorarea unei soluţii**

  `moneda[s]` = indicele **i** pentru care se realizează minimul din formula pentru **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 ≤ 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 (vi ≤ s && nr[s − vi] + 1 < nr[s]){
                  nr[s] = nr[s − vi] + 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:

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

și

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

strategia Greedy furnizează soluția optimă.

Descompunerea unui dreptunghi în pătrate

- Se consideră un dreptunghi cu laturile de m, respectiv n unităţi (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 x 6 poate fi descompus în două pătrare 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 şi 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**

$$a[i][1]=i,\ \forall i=1,...,m$$
$$a[1][j]=j,\ \forall j=1,...,n$$
$$a[i][i]=1,\ \forall =1,...,m$$

- **Relație de recurență**

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

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

$a[j][i] = a[i][j]$, pentru $j \le m$

- **Ordinea de parcurgere a grafului de dependențe (ordinea de calcul)**

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

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 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

- Generararea tuturor elementelor produsului cartezian X nu este acceptabilă.

  Metoda backtracking încearcă micşorarea timpului de calcul.

Metoda Backtracking

- Vectorul x este **construit progresiv**, începând cu prima componentă.
- Se avansează cu o valoare x_k dacă este satisfăcută **condiţia 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

- Cazuri posibile la alegerea lui x_k:
  - **Atribuie şi avansează**
  - **Încercare eşuată**
  - **Revenire**
  - **Revenire după determinarea unei soluţii**

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

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

$k \leftarrow 1$;

- 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 $\mathbf{X_i} = \{\mathbf{p_i}, \mathbf{p_i+1}, \ldots, \mathbf{u_i}\}$ 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;
```

- 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 $\mathbf{X_i = \{p_i, p_i+1, \ldots, u_i\}}$ 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 }
       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
  else
      for (i=pk; i<=uk; i++) valori posibile
          xk←i;
          if φk(x1,…,xk)
              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 condiţiile de continuare $\varphi_k(x_1, \ldots, x_k)$ vom folosi funcţia `cont(k)`

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 (adică să nu existe două dame pe aceeaşi linie, coloană sau diagonală).

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)**

  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 $|x_i - x_k| \neq k - i$

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 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)**

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

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

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

Şiruri 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**

Şiruri corecte de paranteze

- **Reprezentarea soluţiei**

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

  $\mathbf{x}_k \in \{0,1\}$

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

- **Condiţii interne (finale)**

- **Condiţii de continuare**

Şiruri corecte de paranteze

- **Reprezentarea soluţiei**

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

  $\mathbf{x}_k \in \{0,1\}$

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

- **Condiţii interne (finale)**

  $\mathbf{dif}=0$

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

- **Condiţii de continuare**

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

Şiruri corecte de paranteze

- **Reprezentarea soluţiei**

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

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

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

- **Condiţii interne (finale)**

  $\mathbf{dif=0}$

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

- **Condiţii de continuare**

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

  $\mathbf{dif} \leq \mathbf{n-k}$ -> şi 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:
  - soluţia poate avea un număr variabil de componente *şi/sau*
  - dintre ele alegem una care optimizează o funcţie dată

- **Exemplu**: Fie $a=(a_1,\ldots,a_n)\in\mathbf{Z}^n$. Să se determine un subşir crescător de lungime maximă.

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

- **Reprezentarea soluţiei**

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

  $x_k \in \{1, \dots, n\}$

- **Condiţii interne (finale)**

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

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

- **Condiţii de continuare**

  $x_{k-1} \le x_k \longrightarrow x_k \in \{x_{k-1}, \dots, n\}$

  $x_1 + x_2 + \dots + x_k \le n$

```
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>=1){
        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 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 Est, Nord, Vest şi 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 ieşirea din matrice.

Backtracking în plan

- Dacă poziţia este liberă şi putem continua, setăm $a_{ij}=-1$ (a fost atinsă), continuăm şi apoi repunem $a_{ij}\leftarrow 0$ (întoarcere din recursivitate).
- Pentru refacerea drumurilor reținem un vector cu pozițiile parcurse sau pentru fiecare poziţie 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
                    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)
```