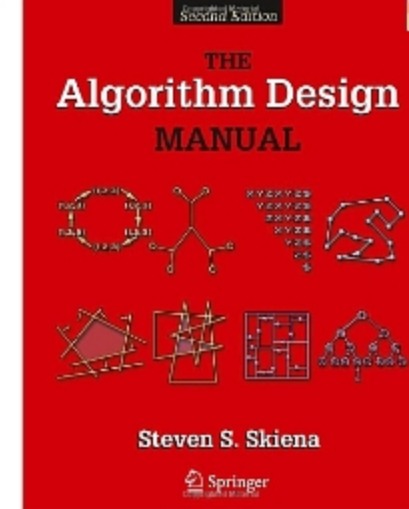


# CURS 9:

## Tehnici de proiectare a algoritmilor - Tehnica reducerii-

# Motivație



## Interview Problems

- 7-14. [4] Write a function to find all permutations of the letters in a particular string.
- 7-15. [4] Implement an efficient algorithm for listing all  $k$ -element subsets of  $n$  items.
- 7-16. [5] An anagram is a rearrangement of the letters in a given string into a sequence of dictionary words, like *Steven Skiena* into *Vainest Knees*. Propose an algorithm to construct all the anagrams of a given string.

# Structura

- Ce este o tehnică de proiectare a algoritmilor ?
- Tehnica forței brute
- Tehnica reducerii
- Algoritmi recursivi și analiza acestora
- Aplicații ale tehnicii reducerii

# Ce este o tehnică de proiectare a algoritmilor?

- ... este o metodă generală de rezolvare algoritmică a unei clase de probleme
- ... o astfel de tehnică poate fi de regulă aplicată mai multor probleme provenind din diferite domenii de aplicabilitate

# De ce sunt utile astfel de tehnici?

- ... furnizează **idei de start și scheme generale de proiectare a algoritmilor destinați rezolvării unor probleme noi**
- ... reprezintă o **colecție de instrumente** utile pentru aplicații

# Care sunt cele mai utilizate tehnici?

- Tehnica forței brute (brute force)
- Tehnica reducerii (decrease and conquer)
- Tehnica divizării (divide and conquer)
- Tehnica căutării local optimale (greedy search)
- Tehnica programării dinamice (dynamic programming)
- Tehnica căutării cu revenire (backtracking)

# Tehnica forței brute

- ... este o **abordare directă** care rezolvă problema pornind de la enunțul acesteia și eventual prin analiza exhaustivă a spațiului soluțiilor (analiza tuturor configurațiilor posibile)
- ... este **cea mai simplă** (și cea mai intuitivă) cale de a rezolva problema
- ... algoritmi proiectați pe baza tehnicii forței brute **nu sunt întotdeauna eficienți**

# Tehnica forței brute

## Exemplu:

- Calculul lui  $x^n$ ,  $x$  este un număr real iar  $n$  este un număr natural

**Idee:** se pornește de la definiția puterii

$$x^n = x * x * \dots * x \text{ (de } n \text{ ori)}$$

```
Power(x,n)
  p ← 1
  FOR i ← 1,n DO
    p ← p*x
  ENDFOR
  RETURN p
```

## Analiza eficienței

Dim. pb:  $n$

Op. dominantă:  $*$

$T(n) = n$

## Clasa de eficiență

$\mathcal{O}(n)$

**Există algoritm mai eficient ?**



# Tehnica forței brute

## Exemplu:

- Calcul  $n!$ , pentru  $n$  un număr natural ( $n \geq 1$ )

**Idee:** se pornește de la definiția factorialului  $n! = 1 * 2 * \dots * n$

```
Factorial(n)
  f ← 1
  FOR i ← 1,n DO
    f ← f*i
  ENDFOR
  RETURN f
```

## Analiza eficienței

Dim. pb:  $n$

Op. dominantă: \*

$T(n) = n$

Clasa de eficiență

$\mathcal{O}(n)$

Există algoritm mai eficient ?

# Tehnica reducerii

## Idee:

- se folosește legătura dintre soluția unei probleme și soluția unei instanțe de dimensiune mai mică a aceleiași probleme.
- prin reducerea succesivă a dimensiunii problemei se ajunge la o instanță suficient de mică pentru a fi rezolvată direct

## Motivație:

- Pentru **unele probleme** o astfel de abordare conduce la algoritmi mai eficienți decât cei obținuți aplicând tehnica forței brute
- Uneori este mai simplu să se specifice relația dintre soluția problemei de rezolvat și soluția unei probleme de dimensiune mai mică decât să se specifice explicit modul de calcul al soluției

# Tehnica reducerii

**Exemplu.** Considerăm problema calculului puterii  $x^n$  pentru  $n=2^m$ ,  $m \geq 1$

Intrucât

$$x^{2^m} = \begin{cases} x * x & \text{pentru } m=1 \\ x^{2^{(m-1)}} * x^{2^{(m-1)}} & \text{pentru } m > 1 \end{cases}$$

rezultă că  $x^{2^m}$  poate fi calculat după schema de mai jos:

$$p := x * x = x^2$$

$$p := p * p = x^2 * x^2 = x^4$$

$$p := p * p = x^4 * x^4 = x^8$$

....

# Tehnica reducerii

Pas 1:  $p := x * x = x^2 = x^{2^1}$

Pas 2:  $p := p * p = x^2 * x^2 = x^4 = x^{2^2}$

Pas 3:  $p := p * p = x^4 * x^4 = x^8 = x^{2^3}$

...

Pas (m-1):  $p := p * p = x^{2^{m-1}} * x^{2^{m-1}} = x^{2^m}$

```
Power2(x,m)
  p ← x*x
  FOR i ← 1,m-1 DO
    p ← p*p
  ENDFOR
  RETURN p
```

**Obs:** abordarea din Power2 este ascendentă (bottom-up) în sensul că se pornește de la problema de dimensiune mică către problema de dimensiune mare

# Tehnica reducerii

```
Power2(x,m)
  p ← x*x
  FOR i ← 1,m-1 DO
    p ← p*p
  ENDFOR
  RETURN p
```

Analiza :

a) Corectitudine

Invariant ciclu:  $p=x^{2^i}$

b) Eficiența

(i) dimensiune problemă:  $m$

(ii) operație dominantă:  $*$

$$T(m) = m$$

Observație:

$$m=\log(n)$$

# Tehnica reducerii

$$x^{2^m} = \begin{cases} x*x & \text{pentru } m=1 \\ x^{2^{m-1}} * x^{2^{m-1}} & \text{pentru } m>1 \end{cases}$$

$$x^n = \begin{cases} x*x & \text{pentru } n=2 \\ x^{n/2} * x^{n/2} & \text{pentru } n>2 \end{cases}$$

power3(x,m)

IF m=1 THEN RETURN x\*x

ELSE

p ← power3(x,m-1)

RETURN p\*p

ENDIF

dimensiunea descrește  
cu 1

power4(x,n)

IF n=2 THEN RETURN x\*x

ELSE

p ← power4(x,n DIV 2)

RETURN p\*p

ENDIF

Dimensiunea descrește  
prin împărțire la 2

# Tehnica reducerii

power3(x,m)

```
IF m=1 THEN RETURN x*x
ELSE
  p ← power3(x,m-1)
  RETURN p*p
ENDIF
```

power4(x,n)

```
IF n=2 THEN RETURN x*x
ELSE
  p ← power4(x,n DIV 2)
  RETURN p*p
ENDIF
```

Observatii:

1. In algoritmi de mai sus se folosește o abordare descendentă (top-down): se pornește de la problema de dimensiune mare și se reduce succesiv dimensiunea până se ajunge la o problemă suficient de simplă
2. Ambii algoritmi sunt **recursivi**

# Tehnica reducerii

Ideea poate fi extinsă in cazul unui exponent  $n$  cu valoare naturală arbitrară

$$x^n = \begin{cases} x & \text{pentru } n=1 \\ x^{n/2} * x^{n/2} & \text{pentru } n \geq 2, n \text{ par} \\ x^{(n-1)/2} * x^{(n-1)/2} * x & \text{pentru } n > 2, n \text{ impar} \end{cases}$$

```
power5(x,n)
```

```
  IF n=1 THEN RETURN x
```

```
  ELSE
```

```
    p ← power5(x,n DIV 2)
```

```
    IF n MOD 2=0 THEN RETURN p*p
```

```
      ELSE RETURN p*p*x
```

```
  ENDIF
```

```
ENDIF
```



# Structura

- Ce este o tehnică de proiectare a algoritmilor ?
- Tehnica forței brute
- Tehnica reducerii
- Algoritmi recursivi și analiza acestora
- Aplicații ale tehnicii reducerii

# Algoritmi recursivi

## Noțiuni

- Algoritm recursiv = un algoritm care conține cel puțin un **apel recursiv**
- Apel recursiv = **apelul aceluiași algoritm fie direct** (algoritmul A se autoapelează) fie **indirect** (algoritmul A apelează algoritmul B care apelează la rândul lui algoritmul A)

## Observații:

- Cascada apelurilor recursive este echivalentă cu un proces iterativ
- Un algoritm recursiv trebuie să conțină un caz particular pentru care se poate returna direct rezultatul fără să fie necesar apelul recursiv
- Algoritmii recursivi sunt ușor de implementat dar execuția apelurilor recursive induce costuri suplimentare (la fiecare apel recursiv se plasează o serie de informații într-o zonă de memorie organizată ca o **stivă** - numită chiar stiva programului)

# Exemplu

## Calcul factorial

$$n! = \begin{cases} 1 & n \leq 1 \\ (n-1)! * n & n > 1 \end{cases}$$

```
fact(n)
  If n<=1 then rez ← 1
                else rez ← n*fact(n-1)
  endif
return rez
```

# Algoritmi recursivi – mecanism de apel

fact(4): Stiva = [4]

fact(3): Stiva = [3,4]

fact(2): Stiva = [2,3,4]

fact(1): Stiva = [1,2,3,4]

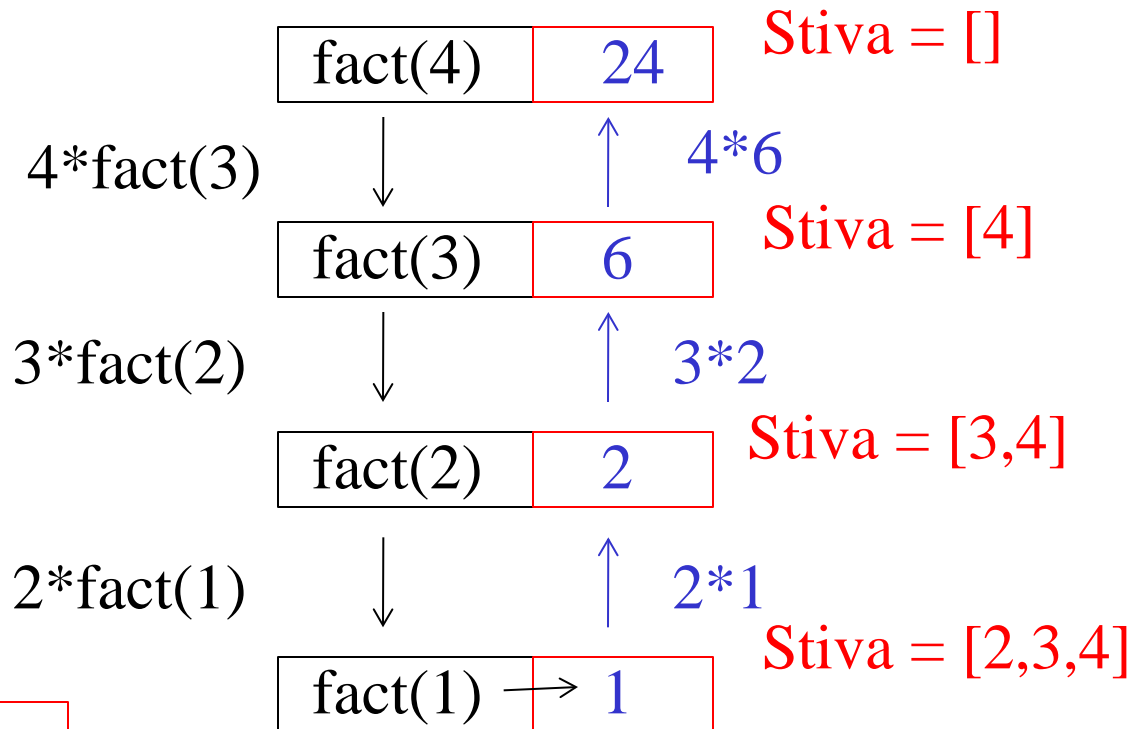
fact(n)

    If  $n \leq 1$  then rez  $\leftarrow$  1

        else rez  $\leftarrow$   $n \cdot \text{fact}(n-1)$

    endif

return rez



Apel  
recursiv

Revenire  
din apel

# Algoritmi recursivi - corectitudine

Intrucât algoritmi recursivi conțin prelucrări iterative (chiar dacă implicite) pentru verificarea corectitudinii este suficient să se identifice o proprietate referitoare la starea algoritmului (similară unui **invariant**) care are proprietățile:

- Este adevărată pentru cazul particular
- Rămâne adevărată după apelul recursiv
- Pentru valorile parametrilor specificate la apelul inițial proprietatea invariantă implică postcondiția

**Exemplu:** (calcul factorial). Proprietatea satisfăcută la orice apel **rez=n!** (unde  $n$  este valoarea curentă a parametrului)

Caz particular:  $n=1 \Rightarrow rez=1=n!$

Dupa execuția apelului recursiv **rez=(n-1)!\*n=n!**

# Algoritmi recursivi - corectitudine

Exemplu. P: a,b nr naturale, a,b≠0; Q: returnează cmmdc(a,b)

Relația de recurență specifică cmmdc:

$$\text{cmmdc}(a,b) = \begin{cases} a & \text{dacă } b=0 \\ \text{cmmdc}(b, a \text{ MOD } b) & \text{dacă } b \neq 0 \end{cases}$$

cmmdc(a,b)

IF b=0 THEN rez ← a

ELSE rez ← cmmdc(b, a MOD b)

ENDIF

RETURN rez

Invariant: rez=cmmdc(a,b)

Caz particular: b=0 => rez=a=cmmdc(a,b)

Dupa apelul recursiv: pentru b ≠ 0

cmmdc(a,b)=cmmdc(b, a MOD b)

rezultă că rez=cmmdc(a,b)

Pentru valorile de apel ale parametrilor:

rez=cmmdc(a,b) => Q

# Algoritmi recursivi – eficiența

## Etapele analizei eficienței:

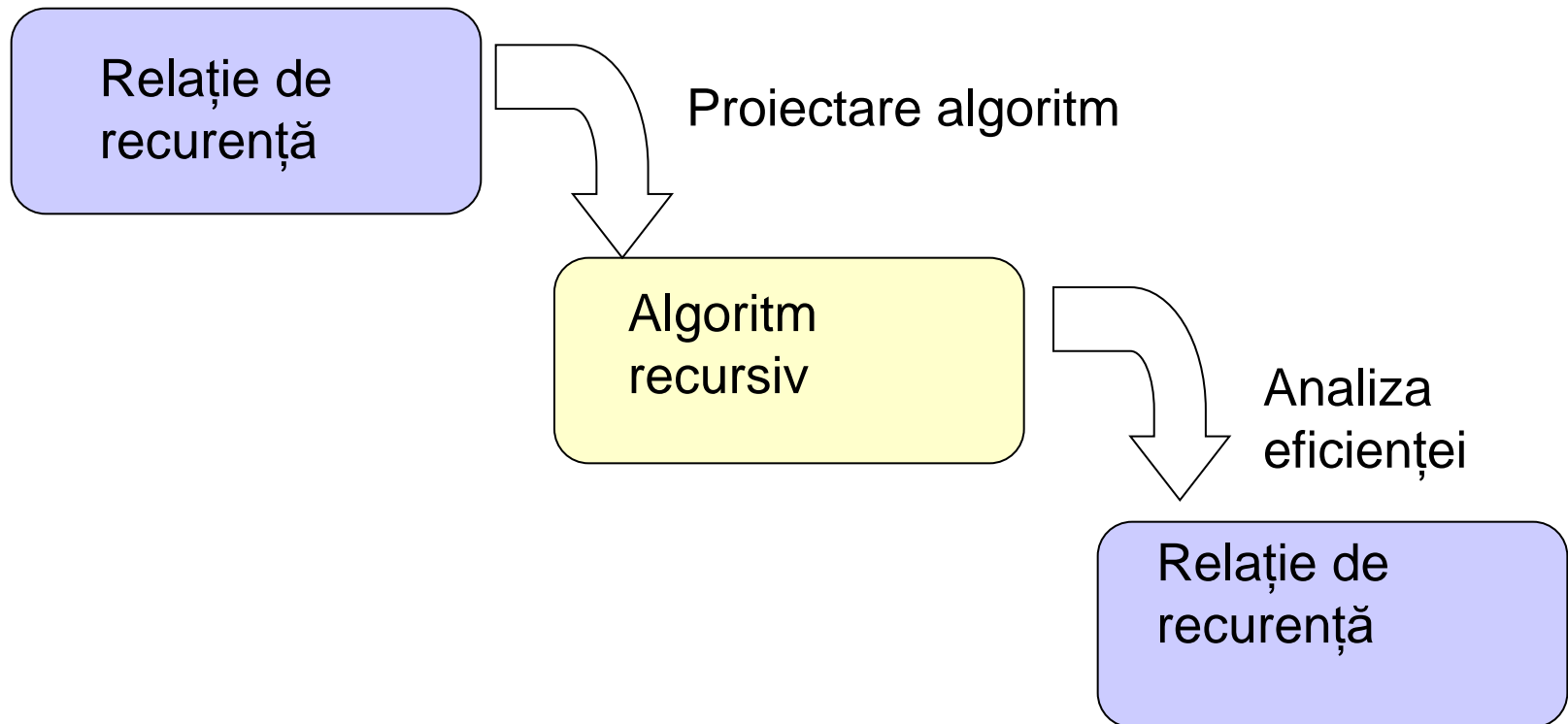
- Stabilirea dimensiunii problemei
- Alegerea operației dominante
- Se verifică dacă timpul de execuție depinde și de proprietățile datelor de intrare (în această situație se analizează cazul cel mai favorabil și cazul cel mai defavorabil)
- Estimarea timpului de execuție

În cazul algoritmilor recursivi pentru estimarea timpului de execuție se stabilește **relația de recurență care exprimă legătura dintre timpul de execuție corespunzător problemei inițiale și timpul de execuție corespunzător problemei reduse** (de dimensiune mai mică)

Estimarea timpului de execuție se obține prin rezolvarea relației de recurență

# Algoritmi recursivi – eficiența

Observație:





# Algoritmi recursivi – eficiența

```
rec_alg (n)
  IF n=n0 THEN <P>
    ELSE rec_alg(h(n))
  ENDIF
```

Ipoteze:

- <P> este prelucrarea corespunzătoare cazului particular și este de cost  $c_0$
- $h$  este o funcție descrescătoare și există  $k$  astfel încât  
 $h^{(k)}(n)=h(h(\dots(h(n))\dots))=n_0$
- Costul calculului lui  $h(n)$  este  $c$

Cu aceste ipoteze relația de recurență pentru timpul de execuție poate fi scrisă:

$$T(n) = \begin{cases} c_0 & \text{dacă } n=n_0 \\ T(h(n))+c & \text{dacă } n>n_0 \end{cases}$$

# Algoritmi recursivi – eficiența

Calcul  $n!$ ,  $n \geq 1$

Relația de recurență:

$$n! = \begin{cases} 1 & n=1 \\ (n-1)! * n & n>1 \end{cases}$$

Algoritm:

fact(n)

IF  $n \leq 1$  THEN RETURN 1

ELSE RETURN fact(n-1)\*n

ENDIF

Dimensiune problemă:  $n$

Operație dominantă: înmulțirea

Relația de recurență pentru timpul de execuție:

$$T(n) = \begin{cases} 0 & n=1 \\ T(n-1)+1 & n>1 \end{cases}$$

# Algoritmi recursivi – eficiența

Metode de rezolvare a relațiilor de recurență:

- Substituție directă
  - Se porneste de la cazul particular și se construiesc termeni succesivi folosind relația de recurență
  - Se identifică forma termenului general
  - Se verifică prin calcul direct sau prin inducție matematică expresia timpului de execuție
- Substituție inversă
  - Se pornește de la cazul  $T(n)$  și se înlocuiește  $T(h(n))$  cu membrul drept al relației corespunzătoare, apoi se înlocuiește  $T(h(h(n)))$  și așa mai departe, până se ajunge la cazul particular; sau se înmulțesc egalitățile cu factori care să permită eliminarea tuturor termenilor de forma  $T(h(n))$  cu excepția lui  $T(n)$
  - Se efectuează calculele și se obține  $T(n)$

# Algoritmi recursivi – eficiența

Exemplu:  $n!$

$$T(n) = \begin{cases} 0 & n=1 \\ T(n-1)+1 & n>1 \end{cases}$$

Substituție directă

$$T(1)=0$$

$$T(2)=1$$

$$T(3)=2$$

....

$$T(n)=n-1$$

Substituție inversă

$$T(n) = T(n-1)+1$$

$$T(n-1) = T(n-2)+1$$

....

$$T(2) = T(1)+1$$

$$T(1) = 0$$

----- (prin adunare)

$$T(n)=n-1$$

**Obs:** aceeași eficiență ca și algoritmul bazat pe metoda forței brute!

# Algoritmi recursivi – eficiența

Exemplu:  $x^n$ ,  $n=2^m$ ,

power4(x,n)

```
IF n=2 THEN RETURN x*x
ELSE
  p ← power4(x,n/2)
  RETURN p*p
ENDIF
```

$$T(n) = \begin{cases} 1 & n=2 \\ T(n/2)+1 & n>2 \end{cases}$$

$$T(2^m) = T(2^{m-1})+1$$

$$T(2^{m-1}) = T(2^{m-2})+1$$

.....

$$T(2) = 1$$

----- (prin adunare)

$$T(n) = m = \log(n)$$



# Structura

- Ce este o tehnica de proiectare a algoritmilor ?
- Tehnica fortei brute
- Tehnica reducerii
- Algoritmi recursivi si analiza acestora
- Aplicații ale tehnicii reducerii

# Aplicații ale tehnicii reducerii

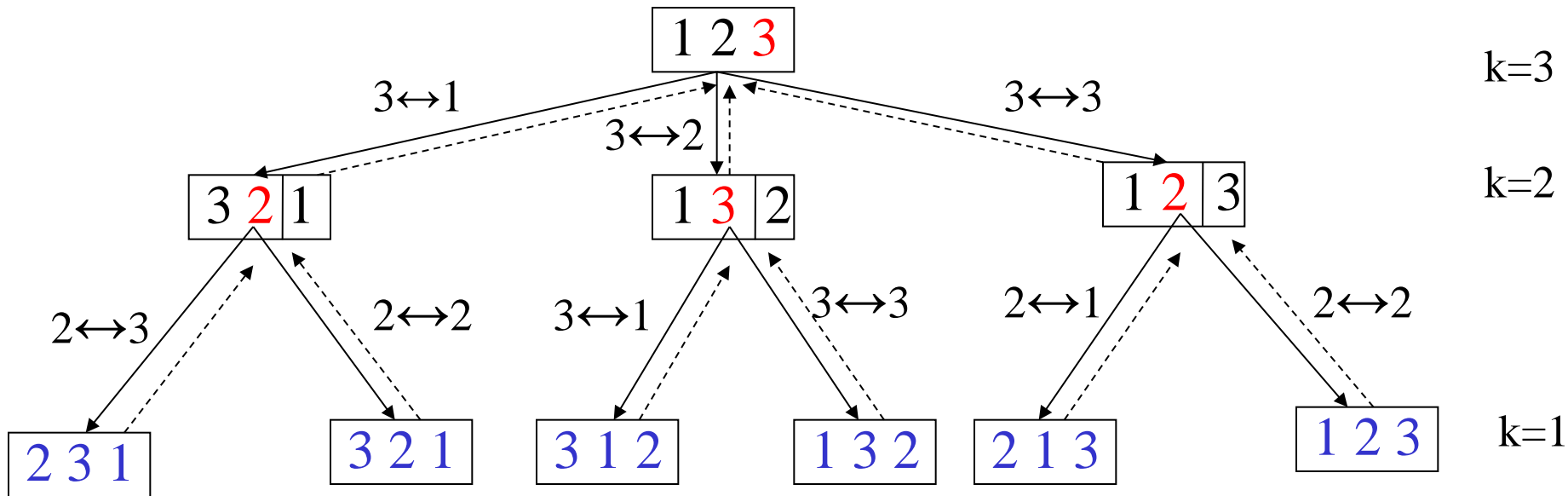
**Exemplu 1:** generarea celor  $n!$  permutări ale mulțimii  $\{1,2,\dots,n\}$

**Idee:** cele  $k!$  permutări ale lui  $\{1,2,\dots,k\}$  pot fi obținute din cele  $(k-1)!$  permutări ale lui  $\{1,2,\dots,k-1\}$  prin plasarea celui de al  $k$ -lea element succesiv pe prima, a doua ... a  $k$ -a poziție. Plasarea lui  $k$  pe poziția  $i$  este realizată prin interschimbarea elementului de pe poziția  $k$  cu cel de pe poziția  $i$ .

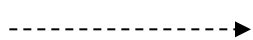


# Generarea permutarilor

Ilustrare pentru  $n=3$  (abordare top-down)



Apel recursiv



Revenire din apelul recursiv

# Generarea permutărilor

Fie  $x[1..n]$  o **variabilă globală** (accesibilă din funcție) conținând inițial valorile  $[1,2,\dots,n]$

Algoritmul are parametrul formal  $k$  și este apelat pentru  $k=n$ .

Cazul particular este  $k=1$ , când tabloul  $x$  conține deja o permutare completă ce poate fi prelucrată (de exemplu, afișată)

```
perm(k)
IF k=1 THEN WRITE x[1..n]
ELSE
  FOR i ← 1,k DO
    x[i] ↔ x[k]
    perm(k-1)
    x[i] ↔ x[k]
  ENDFOR
ENDIF
```

**Analiza eficienței:**

Dim pb.:  $k$

Operație dominantă: **interschimbare**

Relație de recurență:

$$T(k) = \begin{cases} 0 & k = 1 \\ k(T(k-1)+2) & k > 1 \end{cases}$$

**Apel alg:**  $\text{perm}(n)$

# Generarea permutărilor

$$T(k) = \begin{cases} 0 & k=1 \\ k(T(k-1)+2) & k>1 \end{cases}$$

$$T(k) = k(T(k-1)+2)$$

$$T(k-1) = (k-1)(T(k-2)+2) \quad | *k$$

$$T(k-2) = (k-2)(T(k-3)+2) \quad | *k*(k-1)$$

...

$$T(2) = 2(T(1)+2) \quad | *k*(k-1)*... *3$$

$$T(1) = 0 \quad | *k*(k-1)*... *3*2$$

---

$$T(k) = 2(k+k(k-1)+k(k-1)(k-2)+\dots+k!) = 2k!(1/(k-1)!+1/(k-2)!+\dots+1/2+1)$$

->  $2e k!$  (pentru valori mari ale lui  $k$ ). Pt  $k=n \Rightarrow T(n) \in \Theta(n!)$

# Problema turnurilor din Hanoi

**Istoric:** problemă propusă de matematicianul Eduard Lucas în 1883

**Ipoteze:**

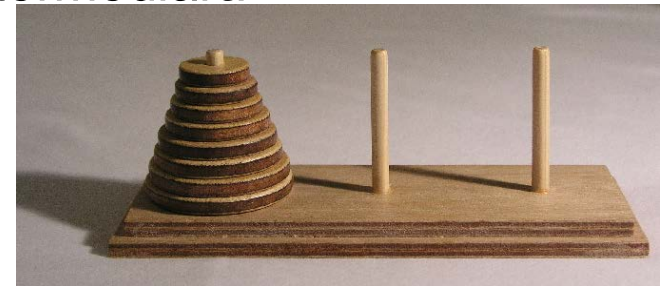
- Considerăm 3 vergele etichetate cu S (sursă), D (destinație) și I (intermediar).
- Inițial pe vergeaua S sunt plasate n discuri de dimensiuni diferite în ordine descrescătoare a dimensiunilor (cel mai mare disc este la baza vergelei iar cel mai mic în varf)

**Scop:**

- Să se mute toate discurile de pe S pe D (la final sunt tot în ordine descrescătoare) utilizând vergeaua I ca intermediară

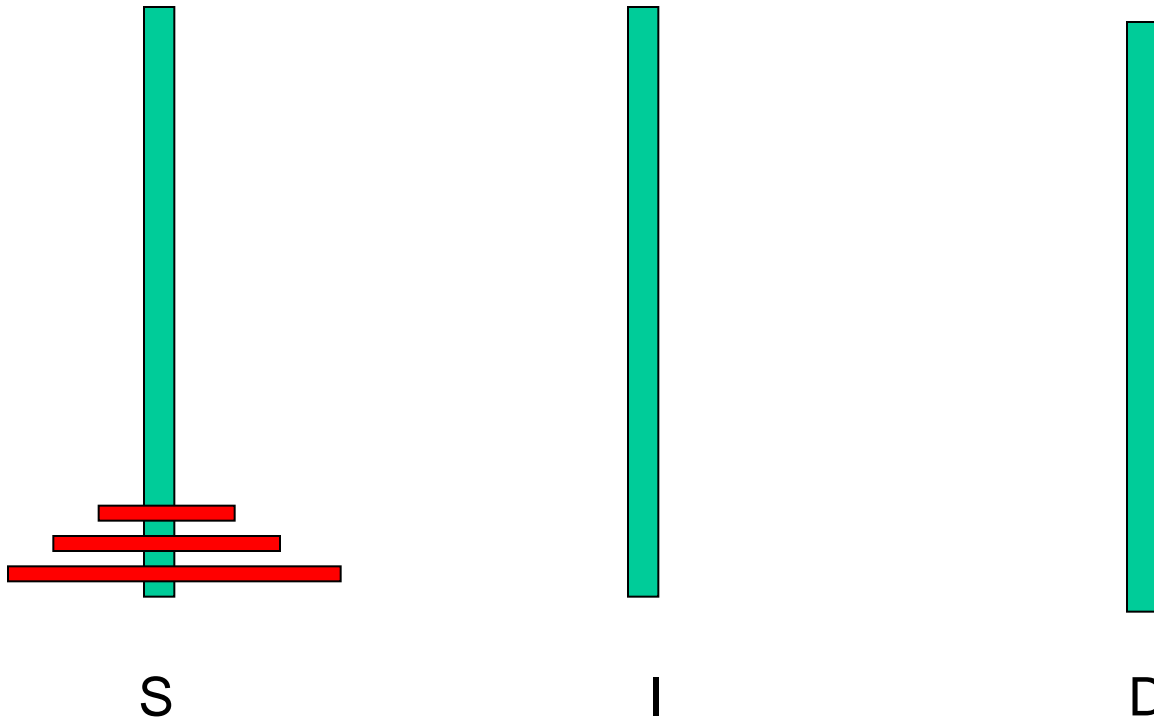
**Restricție:**

- La o etapă se poate muta un singur disc și este interzisă plasarea unui disc mai mare peste un disc mai mic.



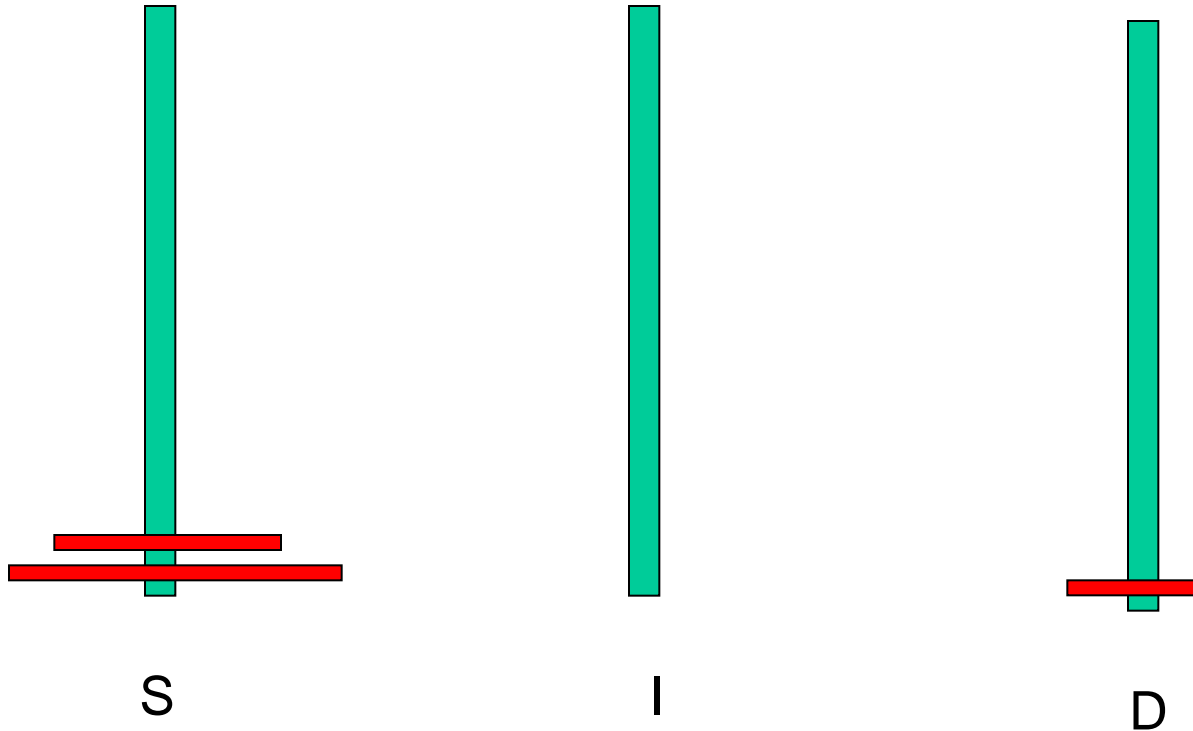
# Problema turnurilor din Hanoi

Prima mutare: S->D



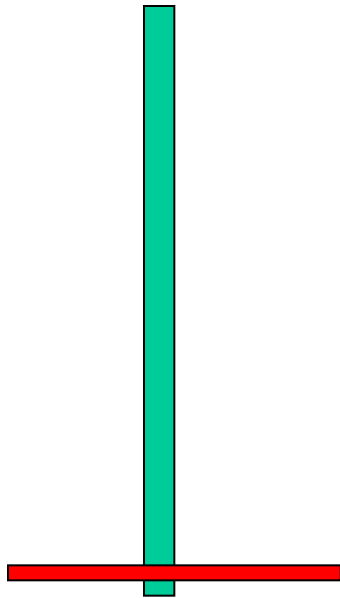
# Problema turnurilor din Hanoi

A doua mutare: S->I

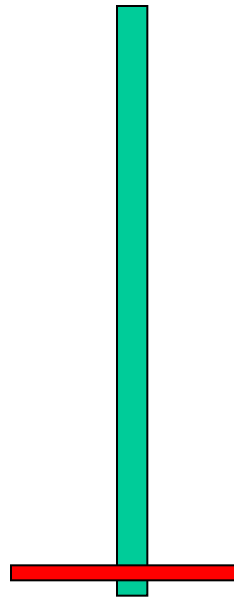


# Problema turnurilor din Hanoi

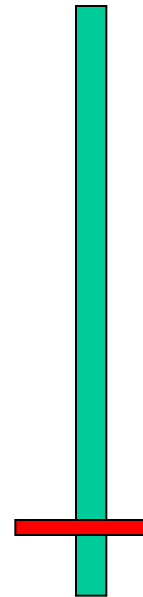
A treia mutare: D->I



S



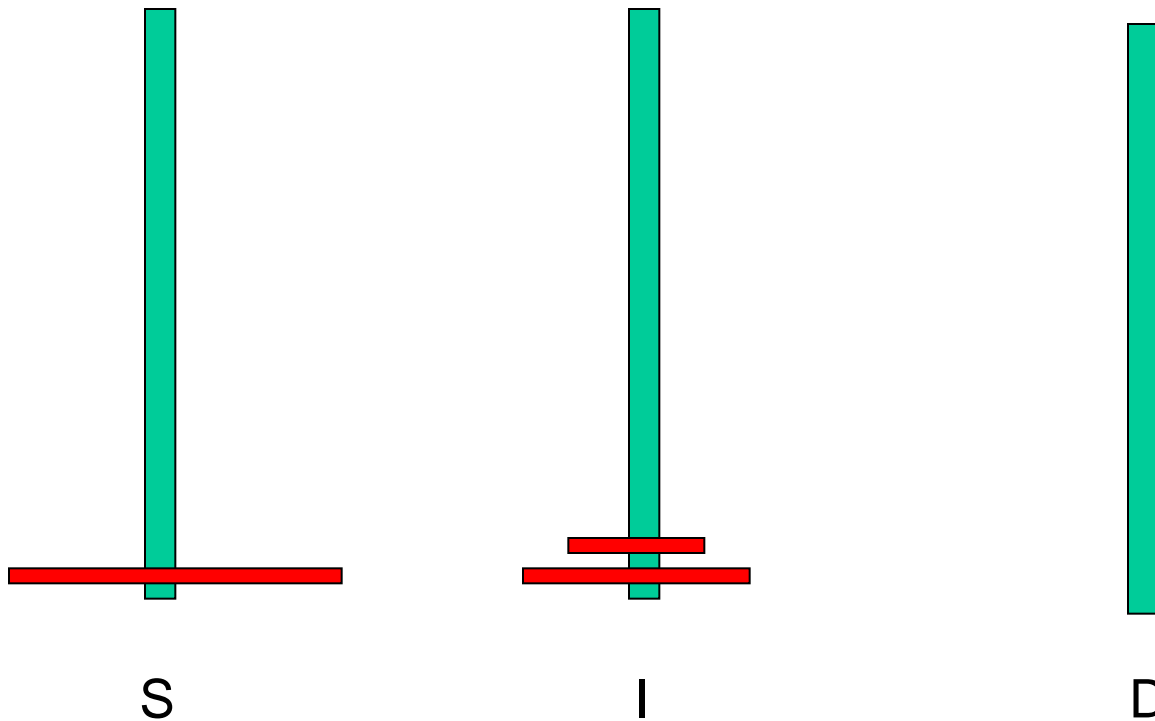
I



D

# Problema turnurilor din Hanoi

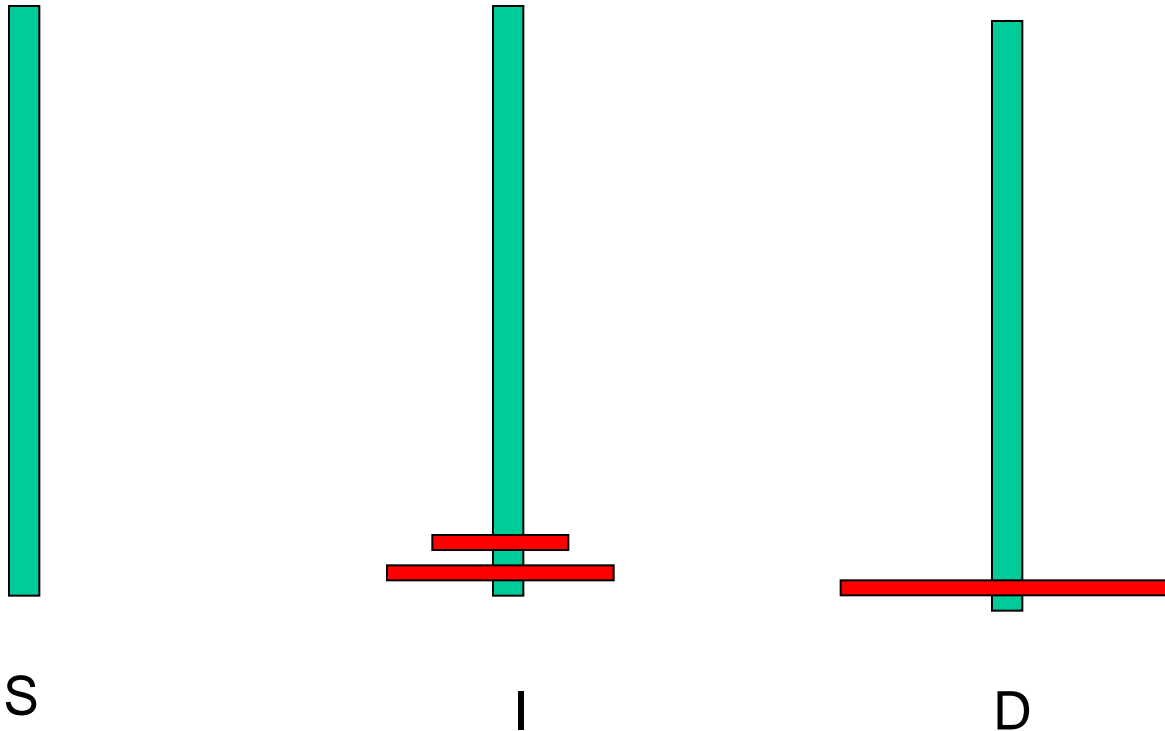
A patra mutare: S->D





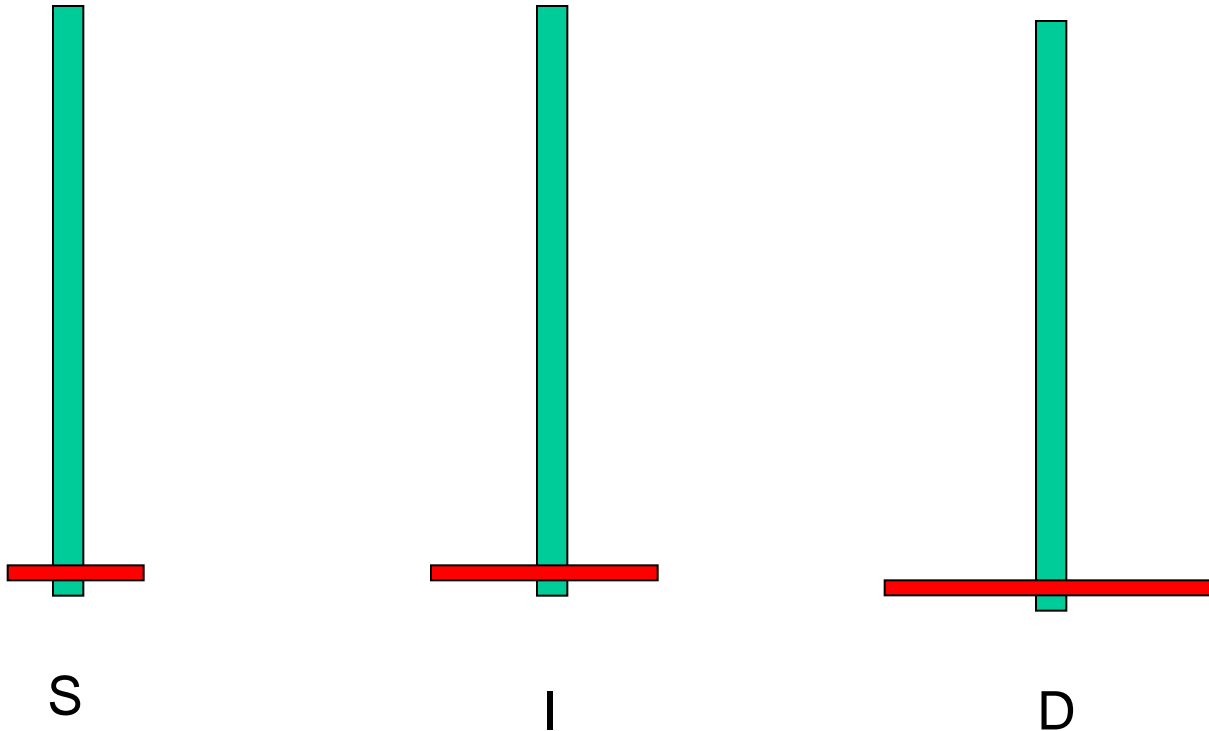
# Problema turnurilor din Hanoi

A cincea mutare: I->S



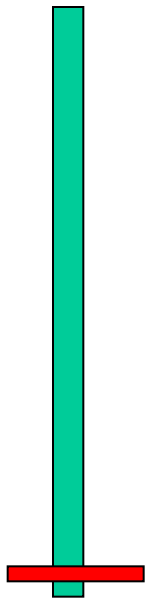
# Problema turnurilor din Hanoi

A sasea mutare: I->D



# Problema turnurilor din Hanoi

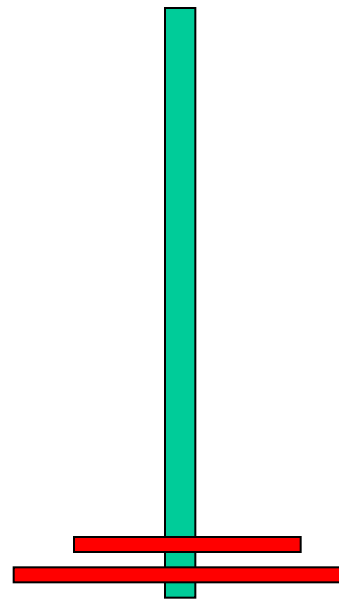
A saptea mutare: S->D



S



I



D

# Problema turnurilor din Hanoi

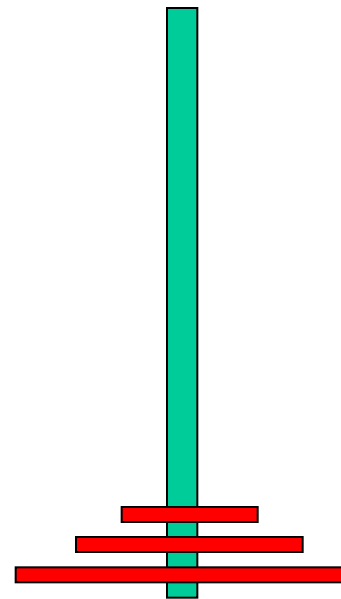
Rezultat



S



I



D

# Problema turnurilor din Hanoi

## Idee:

- Se muta (n-1) discuri de pe S pe I (utilizând D ca vergea auxiliară)
- Se muta discul rămas pe S direct pe D
- Se muta (n-1) discuri de pe I pe D (utilizând S ca vergea auxiliară)

## Algoritm:

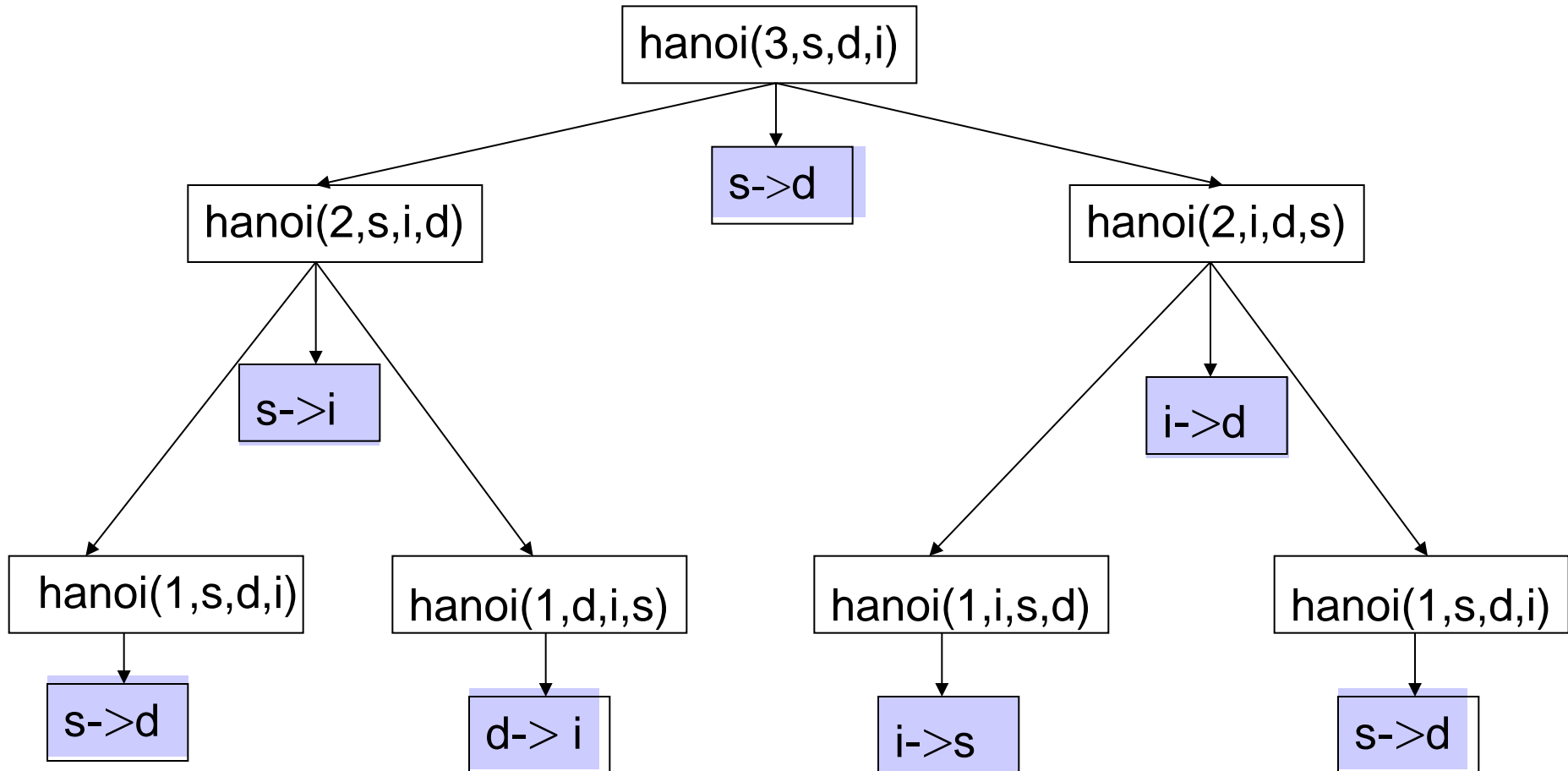
```
hanoi(n,S,D,I)
  IF n=1 THEN  "move from
S to D"
  ELSE hanoi(n-1,S,I,D)
        "move from S to D"
        hanoi(n-1,I,D,S)
  ENDIF
```

## Semnificația parametrilor funcției hanoi:

- Primul parametru: numărul discurilor
- Al doilea parametru: vergea sursă
- Al treilea parametru: vergea destinație
- Al patrulea parametru: vergea intermediară

# Problema turnurilor din Hanoi

Ilustrare apeluri recursive pentru  $n=3$ .



# Problema turnurilor din Hanoi

```
hanoi(n,S,D,I)
  IF n=1 THEN "move from S to D"
  ELSE hanoi(n-1,S,I,D)
        "move from S to D"
        hanoi(n-1,I,D,S)
  ENDIF
```

$$\begin{aligned}T(n) &= 2T(n-1) + 1 \\T(n-1) &= 2T(n-2) + 1 \quad | *2 \\T(n-2) &= 2T(n-3) + 1 \quad | *2^2 \\&\dots \\T(2) &= 2T(1) + 1 \quad | *2^{n-2} \\T(1) &= 1 \quad | *2^{n-1}\end{aligned}$$

---

$$T(n) = 1 + 2 + \dots + 2^{n-1} = 2^n - 1$$

$$T(n) \in \Theta(2^n)$$

Dim pb:  $n$   
Operație dominantă:  $\text{move}$   
Relație de recurență:

$$T(n) = \begin{cases} 1 & n=1 \\ 2T(n-1) + 1 & n > 1 \end{cases}$$

# Variante ale tehnicii reducerii

- Reducere prin scăderea unei constante
  - Exemplu:  $n!$  ( $n!=1$  if  $n=1$   
 $n!=(n-1)!*n$  if  $n>1$ )
- Reducere prin împărțirea la o constantă
  - Exemplu:  $x^n$  ( $x^n=x*x$  if  $n=2$   
 $x^n=x^{n/2}*x^{n/2}$  if  $n>2, n=2^m$ )
- Reducere prin scăderea unei valori variabile
  - Exemplu:  $\text{cmmdc}(a,b)$  ( $\text{cmmdc}(a,b)=a$  pt  $a=b$   
 $\text{cmmdc}(a,b)=\text{cmmdc}(b,a-b)$  pt  $a>b$   
 $\text{cmmdc}(a,b)=\text{cmmdc}(a,b-a)$  pt  $b>a$ )
- Reducere prin împărțire la o valoare variabilă
  - Exemplu:  $\text{cmmdc}(a,b)$   
(  $\text{cmmdc}(a,b)=a$  pt  $b=0$   
 $\text{cmmdc}(a,b)=\text{cmmdc}(b,a \text{ MOD } b)$  pt  $b<>0$ )



# Următorul curs este despre ...

... tehnica divizării

... analiza

.... aplicații

# Întrebare de final

```
alg(n)
if n=0 then rez ← 0
else
  rez ← alg(n DIV 10)+n MOD 10
endif
return rez
```

- a)  $O(n)$
- b)  $\Theta(n)$
- c)  $O(\log(n))$
- d)  $\Theta(\log(n))$
- e)  $\Theta(1)$

Considerând că dimensiunea problemei este  $n$  iar operația dominantă este adunarea, care este ordinul de complexitate al algoritmului?