ALGORITHMS AND DATA STRUCTURES I

Course 7

Organization ...

- Midterm
  - Next week to the seminary

- CCC Contest - www.cloudflight.io
  - Fryday from 16:00

Previous Course

- The problem of sorting
- Sort by insertion
  - Analysis of correctness and efficiency
- Sorting by selection
  - Analysis of correctness and efficiency
- Sorting by interchanging neighboring elements
  - Analysis of correctness and efficiency

Today course

What is an algorithm design technique?

Brute force technique

Reduction technique

Recursive algorithms and their analysis

Applications of the reduction technique

What is an algorithm design technique?

… It is a general method of algorithmic solving of a class of problems

... Such a technique can usually be applied to several problems arising from different fields of applicability

Why are such techniques useful?

… provides starting ideas and general design schemes for algorithms designed to solve new problems

... is a collection of useful app tools

What are the most used techniques?

- Brute force technique
  - **tries all possible solutions** to find the correct or optimal one
- Reduction technique (decrease and conquer)
  - **reduces the problem to a smaller version** of itself, solves that, and uses the solution to solve the original problem
- Division technique (divide and conquer)
  - **divides** a large problem into **two or more smaller subproblems**, solves each independently, and **combines** their results
- Greedy search technique
  - builds a solution **step by step**, always choosing the **best local option** at each stage, hoping it leads to the global optimum
- Dynamic programming technique
  - **breaking them into overlapping subproblems** and **storing their results** to avoid recomputation
- Backtracking technique
  - a **trial-and-error** search method that incrementally builds a solution and **undoes (backtracks)** when a partial solution fails

Today course

What is an algorithm design technique?

Brute force technique

Reduction technique

Recursive algorithms and their analysis

Applications of the reduction technique

Brute force technique

… It is a direct approach that solves the problem starting from its statement and possibly by exhaustively analyzing the solution space (analyzing all possible configurations)

... It's the easiest (and most intuitive) way to solve the problem

... Algorithms designed based on brute force technique are not always efficient

Brute force technique

Example:

Computation of x^n, *x* is a real number and *n* is a natural number

Idea: start from the definition of power

$x^n = x*x*\ldots*x$ (n times)

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

Efficiency analysis

- Pb. dimension: n
- Dominant op: *
- Execution time: $T(n)=n$
- Efficiency class $\Theta(n)$

Is there a more efficient algorithm?

Brute force technique

Example:

Compute n!, for n a natural number (n>=1)

Idea: starting from the definition of factorial n!=1*2*...*n

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

Efficiency analysis

- Pb. dimension: n
- Dominant op.:*
- Execution time: T(n)=n
- Efficiency class: $\Theta(n)$

Is there a more efficient algorithm?

Today course

What is an algorithm design technique?	Brute force technique	Reduction technique	Recursive algorithms and their analysis	Applications of the reduction technique

Reduction technique

Idea

- The link between the solution of a problem and the solution of a smaller instance of the same problem is used.
- Successively reducing the size of the problem results in an instance small enough to be solved directly

Motivation

- For some problems, such an approach leads to more efficient algorithms than those obtained by applying the brute force technique
- Sometimes it is easier to specify the relationship between the solution of the problem to be solved and the solution of a smaller problem than to explicitly specify how to calculate the solution

Reduction technique

Example. Let's consider the problem of computing x at power n, x^n for $n=2^m$, $m>=1$

Whereas

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

x^{2^m} can be calculated according to the scheme below:

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

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

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

….

Reduction technique

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

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

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

…

Step (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
```

Remark: Power2's approach is bottom-up in the sense that it starts from the small problem to the large size problem

Reduction technique

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

Analyze:

a) Correctness

Loop invariant: $p=x^{2^i}$

b) Efficiency

(i) problem size: m

(ii) dominant operation: *

$T(m) = m$

Remark:

$m=\log(n)$

Reduction technique

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

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

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

Size decreases with 1

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

Size decreases by dividing with 2

Reduction technique

```
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
```

Remark:

- In the above algorithms, a top-down approach is used: starting from the large problem and successively reducing the size until a sufficiently simple problem is reached
- Both algorithms are recursive

Reduction technique

The idea can be extended to an exponent, n, of arbitrary natural value

$$x^n = \begin{cases} x & for\ n = 1 \\ x^{n/2} * x^{n/2} & for\ n > 2\ and\ n\ even \\ x^{\frac{n-1}{2}} * x^{\frac{n-1}{2}} * x & for > 2\ and\ n\ odd \end{cases}$$

```
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
```

Today course

- What is an algorithm design technique?
- Brute force technique
- Reduction technique
- Recursive algorithms and their analysis
- Applications of the reduction technique

Recursive algorithms

- Getting
  - Recursive algorithm = an algorithm containing at least one recursive call
  - Recursive call = calling the same algorithm either directly (algorithm A calls itself) or indirectly (algorithm A calls algorithm B, which in turn calls algorithm A)
- Remarks:
  - The cascade of recursive calls is equivalent to an iterative process
  - A recursive algorithm must contain a particular case for which the result can be returned directly without the need for recursive appeal
  - Recursive algorithms are easy to implement, but executing recursive calls incurs additional costs (with each recursive call, a series of information is placed in a memory area organized like a stack - even called the program stack)

Recursive algorithms Example

Factorial computation

Recursive formula:

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

Recursive Algorithm

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

Recursive algorithms
Call mechanism

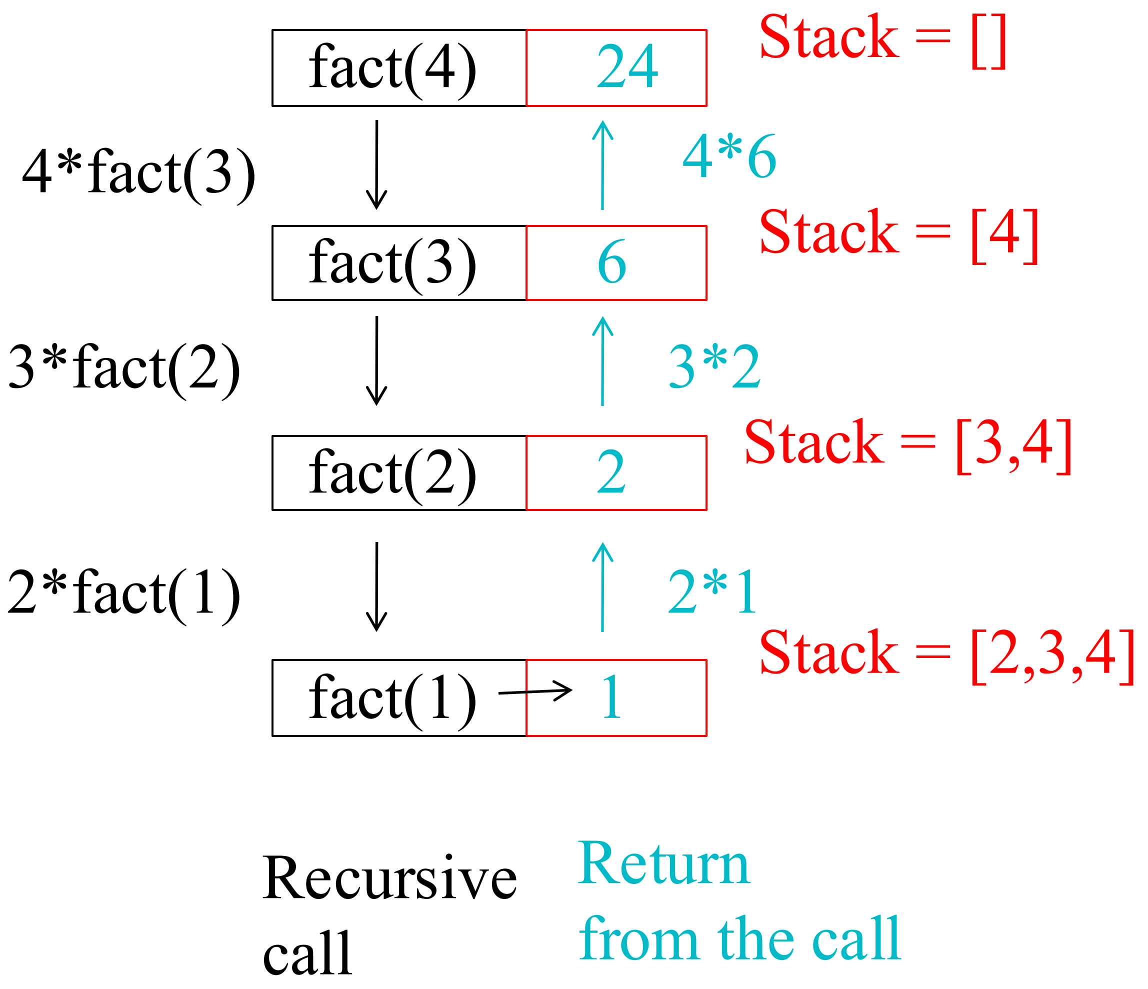
```
fact(n)
    if n<=1 then rez ← 1
    else rez ← n*fact(n-1)
    endif
return rez
```

fact(4): Stack = [4]

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

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

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



Recursive algorithms

- Since recursive algorithms contain an iterative processing, in order to verify the algorithm correctness, it is sufficient to identify an algorithm state property (similar to an invariant) that has the properties:
  - It is true for the particular case
  - Remains true after recursive appeal
  - For parameter values specified on the initial call, the invariant property implies the postcondition

- Example (factorial):
  - Satisfies property on any call $rez=n!$ (where n is the current value of the parameter)
  - Particular case: $n=1 \Rightarrow rez=1=n!$
  - After executing the recursive call $rez=(n-1)!*n=n!$

Recursive algorithms
Correctness

Example. P: a,b numbers, a,b≠0; Q: Returns gcd(a,b)

GCD-specific recurrence relationship:

$$\gcd(a,b) = \begin{cases} a & if\ b = 0 \\ \gcd(b, a\ MOD\ b) & if\ b \neq 0 \end{cases}$$

```
gcd(a,b)
   IF b=0 THEN rez ← a
    ELSE  rez ← gcd(b, a MOD b)
   ENDIF
   RETURN rez
```

Invariant property: rez=gcd(a,b)

Recursive algorithms
Efficiency

Stages of efficiency analysis:

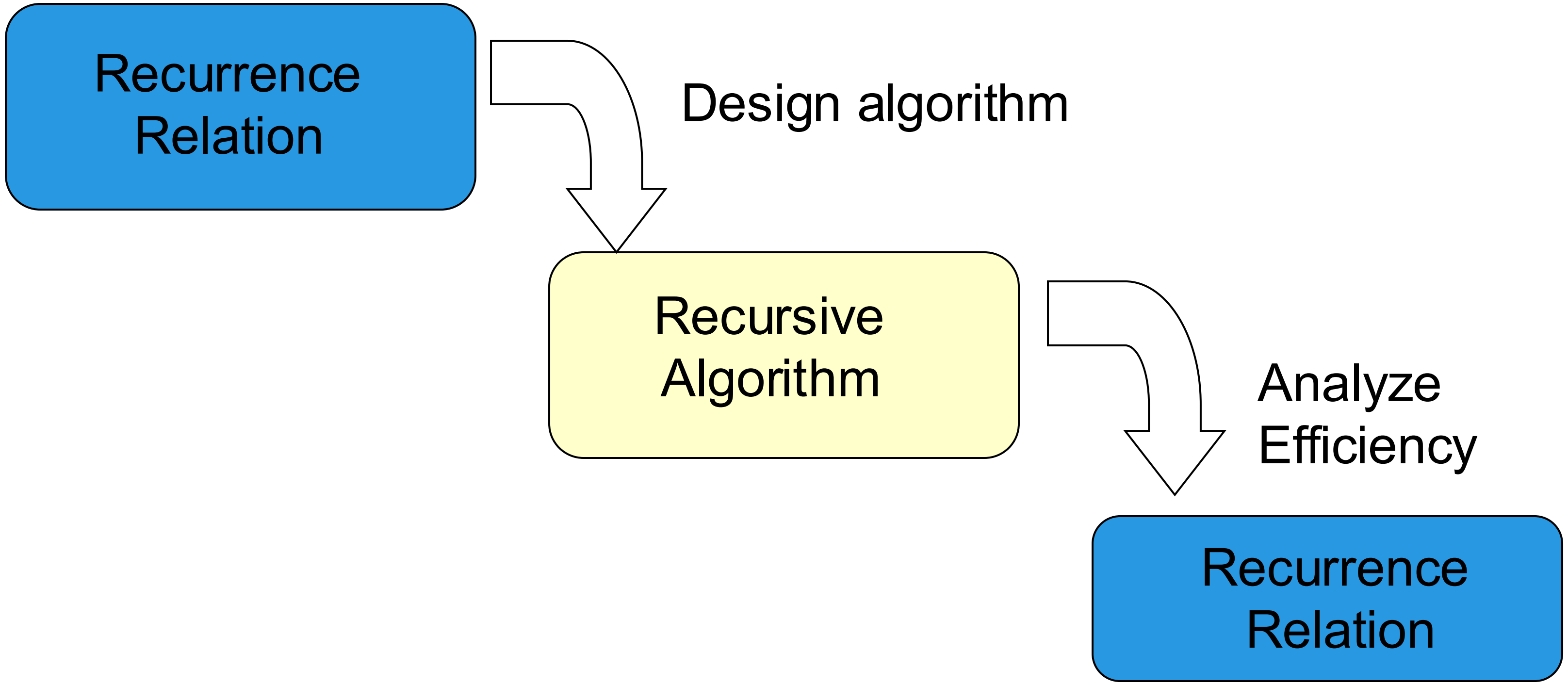
- Determining the size of the problem
- The choice of dominant operation
- Check whether execution time also depends on the properties of the input data (in this case the best case and worst case case are analysed)

Execution time estimation

- In the case of recursive algorithms for estimating execution time, the recurrence relationship is established, expressing the connection between the execution time corresponding to the initial problem and the execution time corresponding to the reduced problem (smaller size)
- The execution time estimation is obtained by solving the recurrence relationship

Recursive algorithms Efficiency

Remark:



Recursive algorithms Efficiency

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

With these assumptions the recurrence relation for execution time can be written:

$$T(n)=\begin{cases} c0 & \text{if } n=n0 \\ T(h(n))+c & \text{if } n>n0 \end{cases}$$

Assumptions:

is the <P> processing corresponding to the particular case and is of cost C0

h is a decreasing function and k exists such that

$h^{(k)}(n)=h(h(\ldots(h(n))\ldots))=n0$

The cost of calculating h(n) is c

Recursive algorithms Efficiency

Computation of n!, n>=1

Recurrence relationship:

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

Algorithm:

```
fact(n)
  IF n<=1 THEN RETURN 1
           ELSE  RETURN fact(n-1)*n
  ENDIF
```

Problem size: n

Dominant operation: multiplication

Recurrence relationship for execution time:

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

Recursive algorithms
Efficiency

Methods for resolving recurrence relationships:

- Direct substitution
  - We start from the particular case and build successive terms using the recurrence relationship
  - The form of the general term is identified
  - The expression of execution time shall be verified by direct calculation or mathematical induction

- Reverse substitution
  - Start from the case T(n) and replace T(h(n)) with the right member of the corresponding relationship, then replace T(h(h(h(n)) and so on, until the particular case is reached; or multiply the equalities by factors allowing the elimination of all terms of the form T(h(n)) except T(n)
  - Perform the calculations and get T(n)

Recursive algorithms Efficiency

Example: n!

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

Direct substitution $T(1)=0$

$T(2)=1$

$T(3)=2$

….

$T(n)=n-1$

Reverse substitution

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

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

….

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

$T(1) \quad =0$

------------------- (by addition)

$T(n)=n-1$

Remark: same efficiency as method-based algorithm brute force!

Recursive algorithms Efficiency

Example: 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
```

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

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

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

….

$T(2) \quad =1$

-------------------- (by addition)

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

Recursive algorithms Efficiency

Remark: in this case the algorithm based on the reduction technique is more efficient than the one based on the brute force method

Explanation: $x^{n/2}$ is calculated only once.

```
pow(x,n)
 IF  n=2 THEN RETURN x*x
 ELSE
   RETURN pow(x,n/2)*pow(x,n/2)
 ENDIF
```

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

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

$T(2^{m-1}) = 2T(2^{m-2})+1 \quad |*2$

$T(2^{m-2}) = 2T(2^{m-3})+1 \quad |*2^2$

….

$T(2) = 1 \quad |*2^{m-1}$

---------------------- (by addition)

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

Today course

- What is an algorithm design technique?
- Brute force technique
- Reduction technique
- Recursive algorithms and their analysis
- Applications of the reduction technique

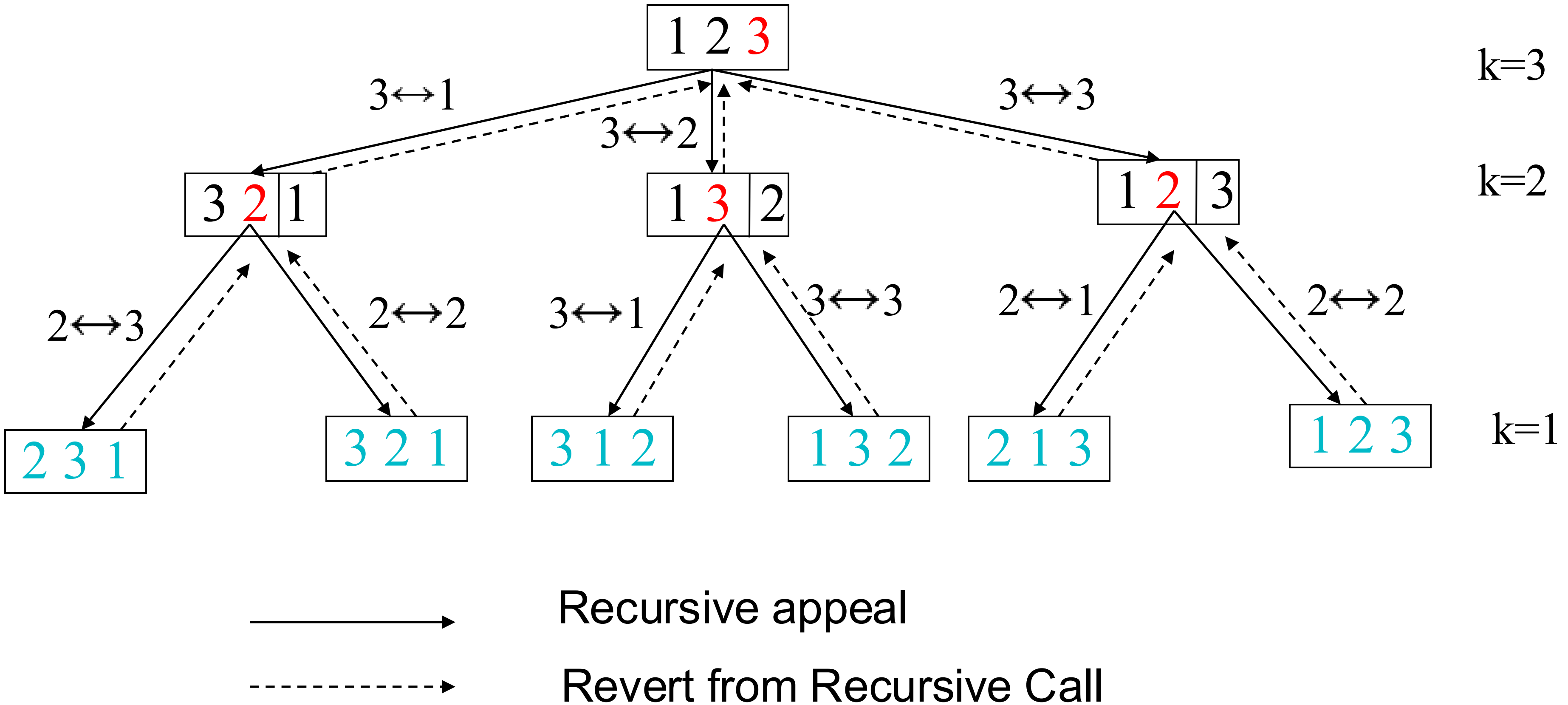
Applications of the reduction technique

Example 1: generating the n! {1,2,...,n} set permutations

Idea: the k! permutations of {1,2,...,k} can be obtained from those (k-1)! permutations of {1,2,...,k-1} by placing the successive k-th element on the first, second ... K-th position. The placement of k at position i is achieved by interchanging the element at position k with that at position i.

Applications of the reduction technique
Generating permutations

Ilustration for n=3 (top-down approach)



Applications of the reduction technique

Generating permutations

Let x[1..n] be a global variable (accessible from function) initially containing the values [1,2,...,n]

The algorithm has the formal parameter k and is called for k=n.

The particular case is k=1, when array x already contains a complete permutation that can be processed (for example, displayed)

```
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
```

Call alg: perm(n)

Efficency analise:

Problem dimension.: k

Dominant operation: swich of 2 values

Recurence relation:

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

Applications of the reduction technique
Generating permutations

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

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

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

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

…

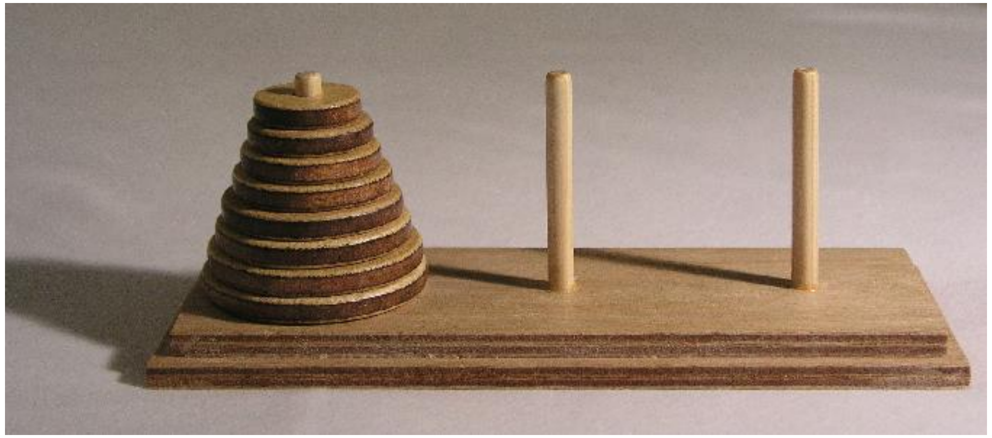
$T(2) \quad =2(T(1)+2) \qquad |*k*(k-1)*\ldots*3$

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

$T(k)=2(k+k(k-1)+ k(k-1)(k-2)+\ldots+k!) =2k!(1/(k-1)!+1/(k-2)!+\ldots+ ½+1)$

$\to$ 2e k! (for large values of k).

Applications of the reduction technique
The problem of towers in Hanoi



- History: problem proposed by mathematician Eduard Lucas in 1883

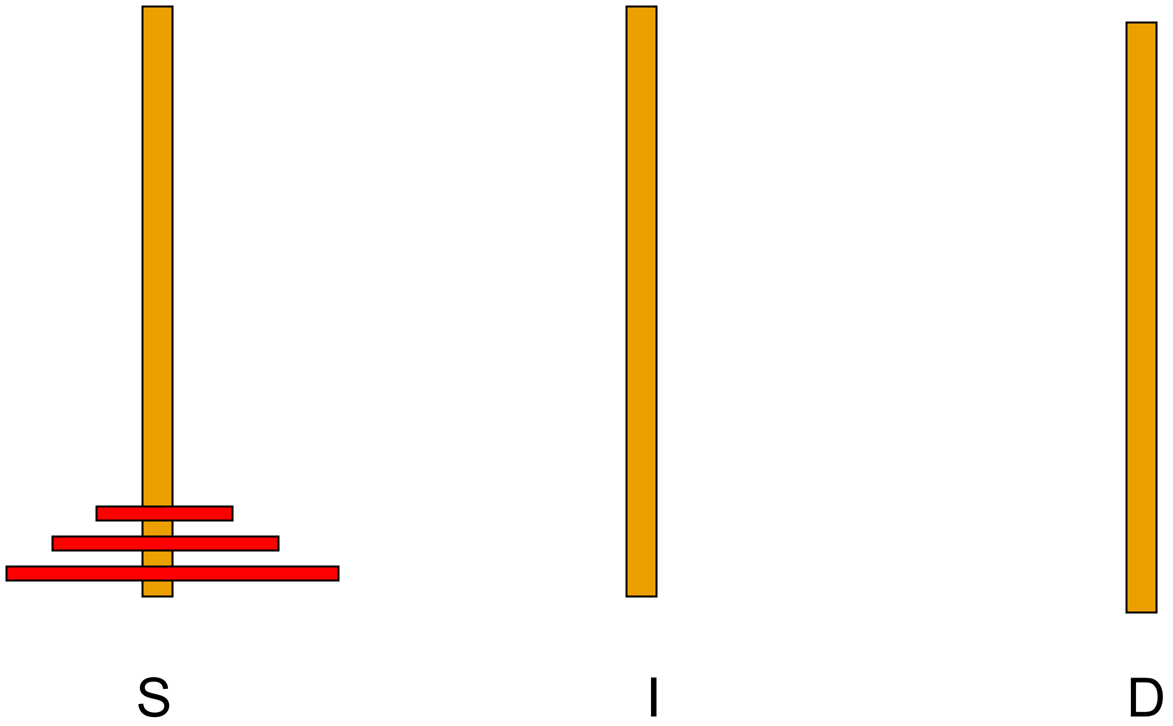
- Assumptions:
  - Consider 3 rods labeled with S (source), D (destination) and I (intermediate).
  - Initially, on the S rod are placed n discs of different sizes in descending order of size (the largest disc is at the base of the rod and the smallest at the tip)

- Purpose:
  - Move all disks from S to D (they are also in descending order at the end) using the I rod as an intermediate

- Restriction:
  - At one stage you can move only one disk and it is forbidden to place a larger disc big over a smaller disk.

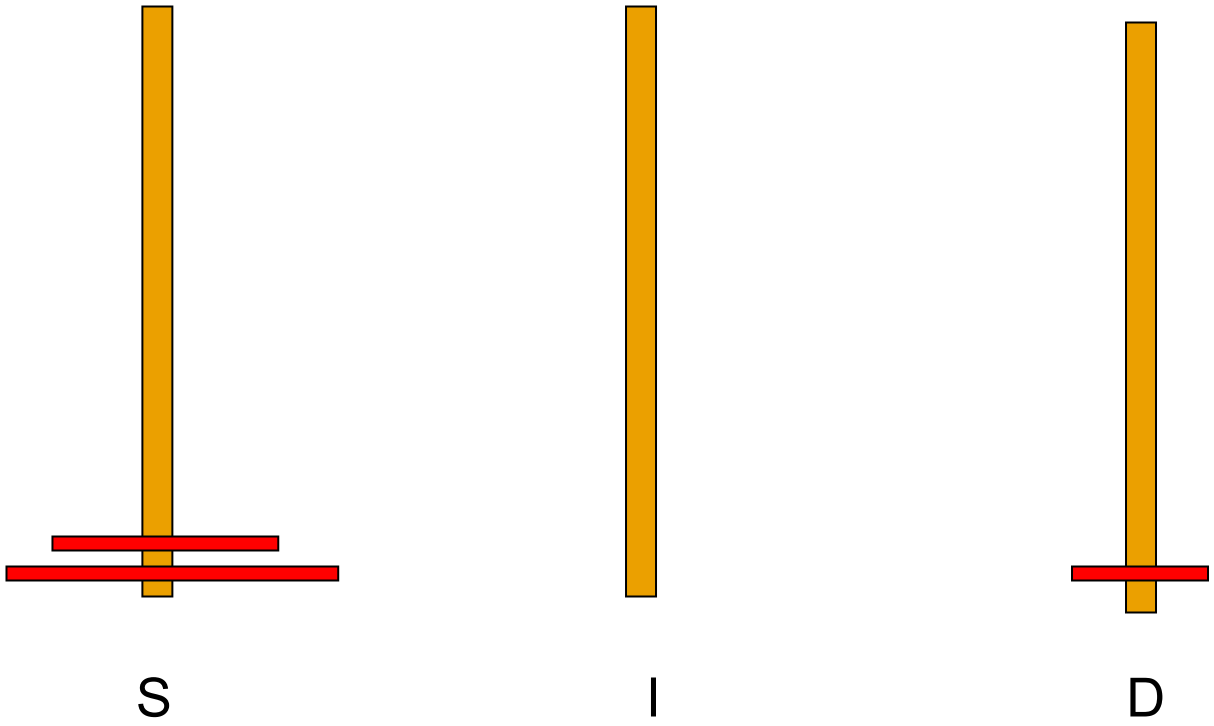
Applications of the reduction technique
The problem of towers in Hanoi

S -> D



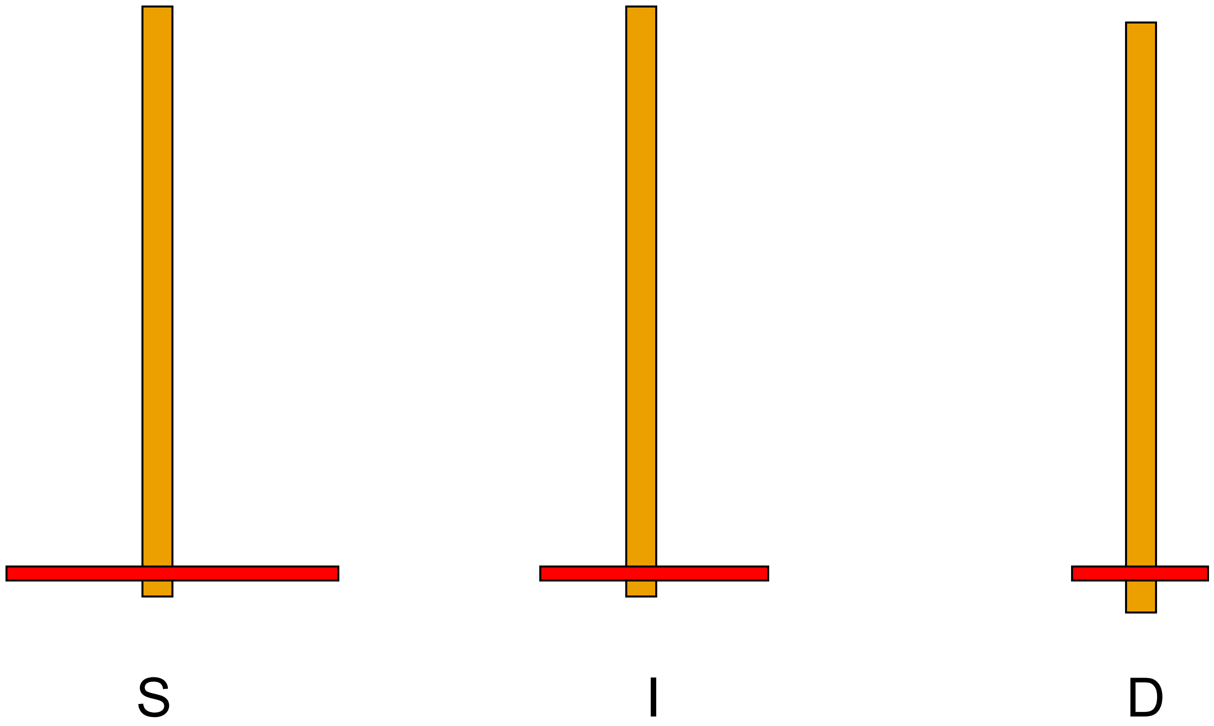
Applications of the reduction technique
The problem of towers in Hanoi

S -> D



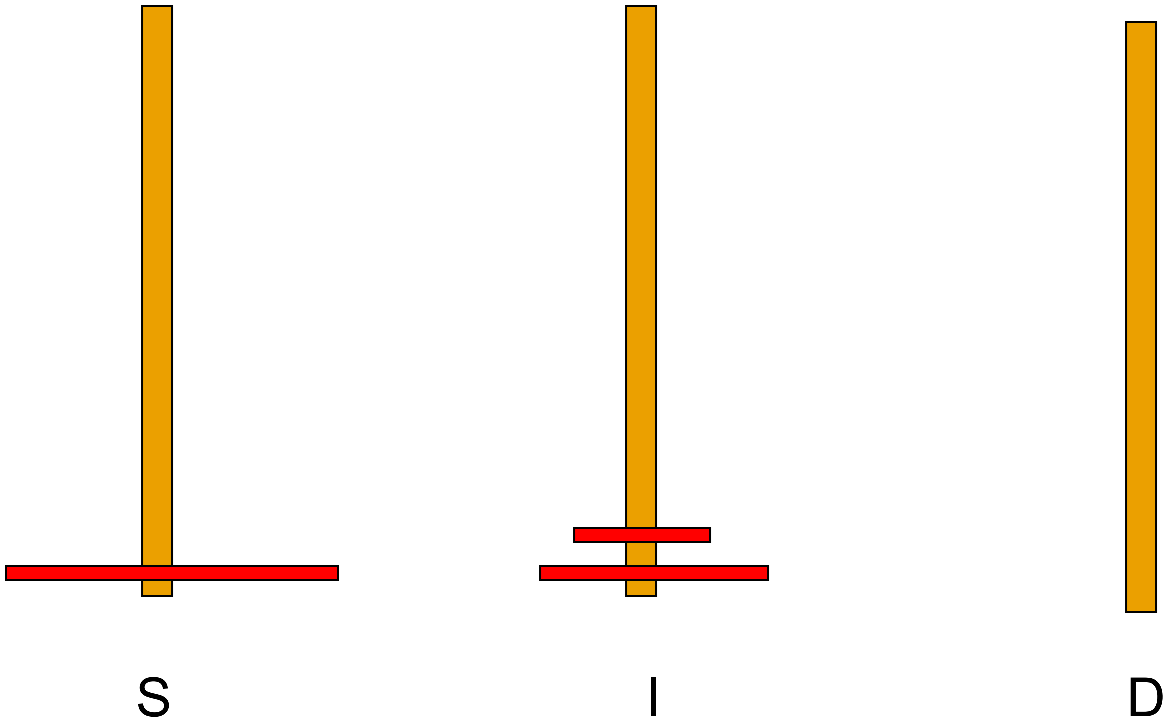
Applications of the reduction technique
The problem of towers in Hanoi

S -> D



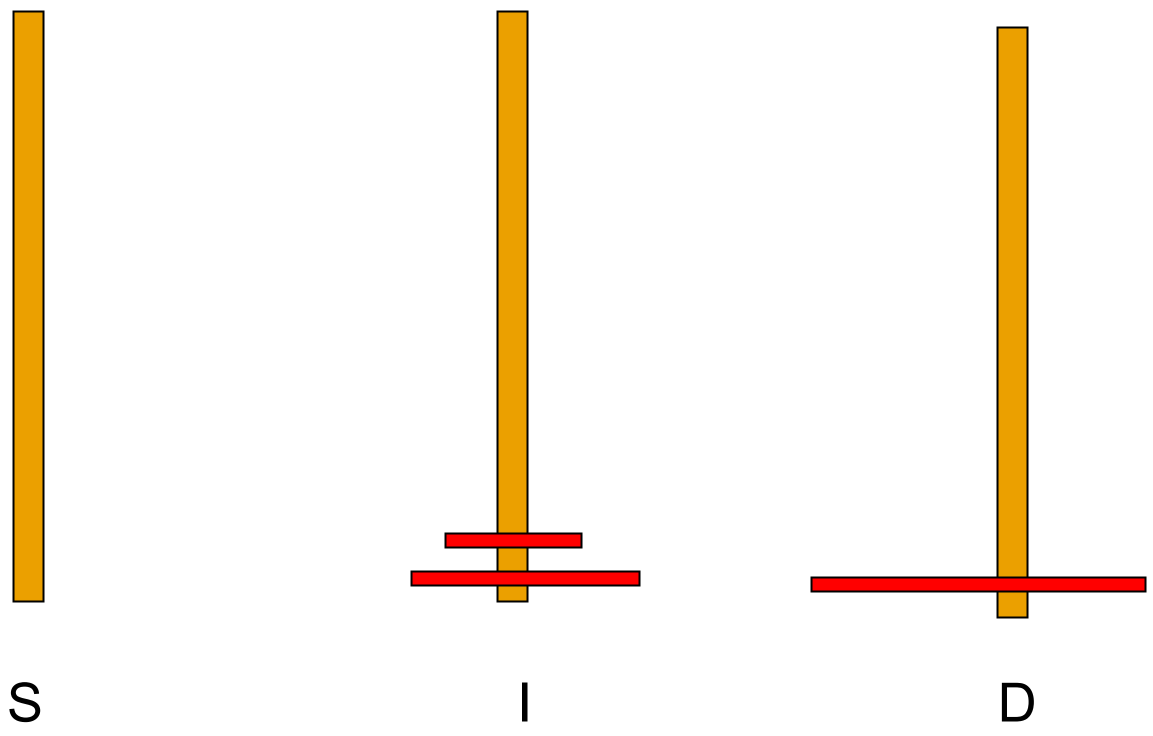
Applications of the reduction technique
The problem of towers in Hanoi

S -> D



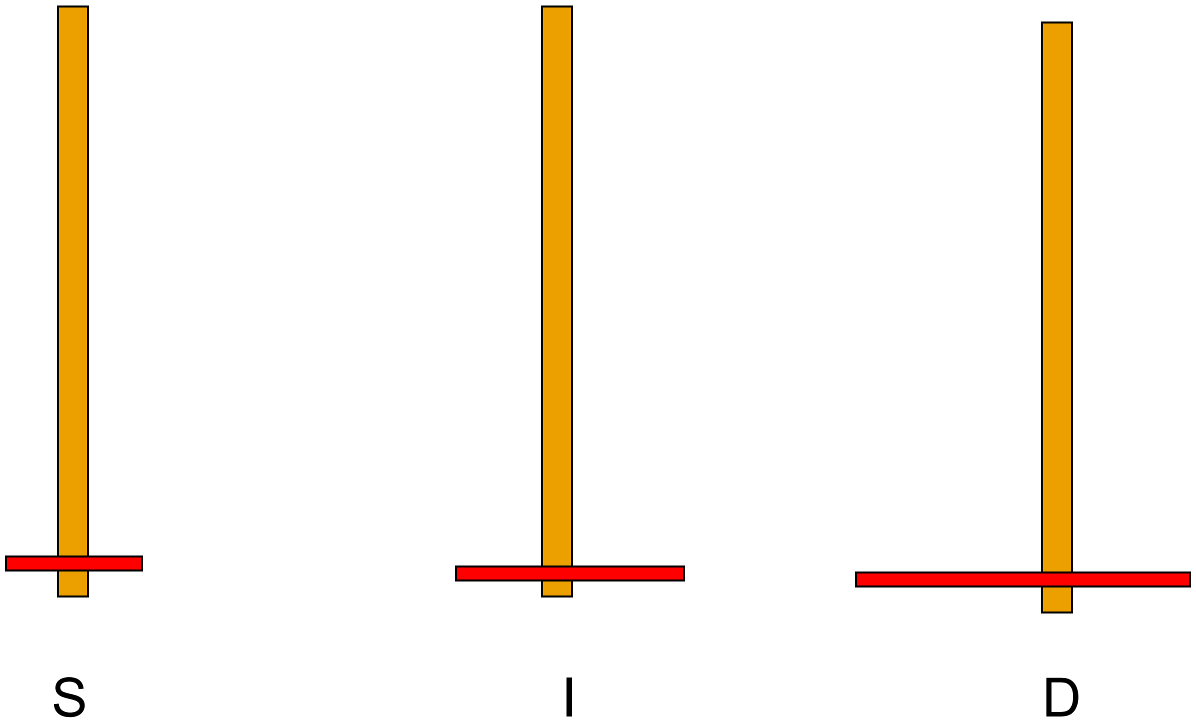
Applications of the reduction technique
The problem of towers in Hanoi

S -> D



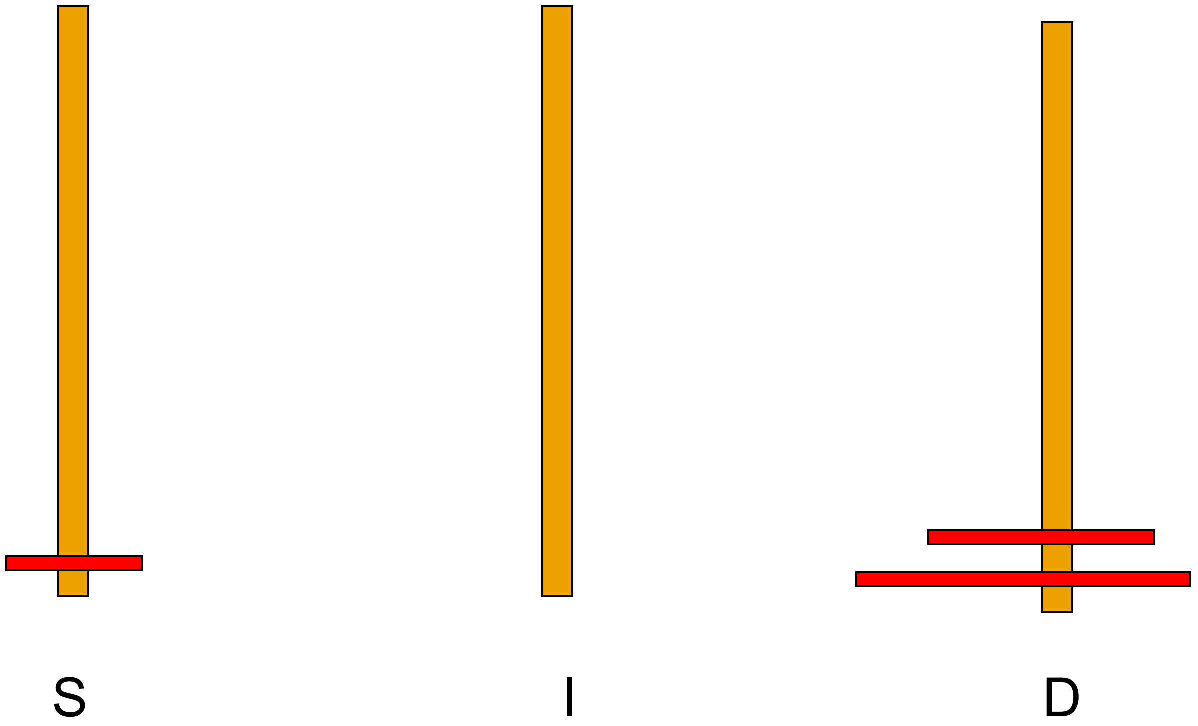
Applications of the reduction technique
The problem of towers in Hanoi

S -> D



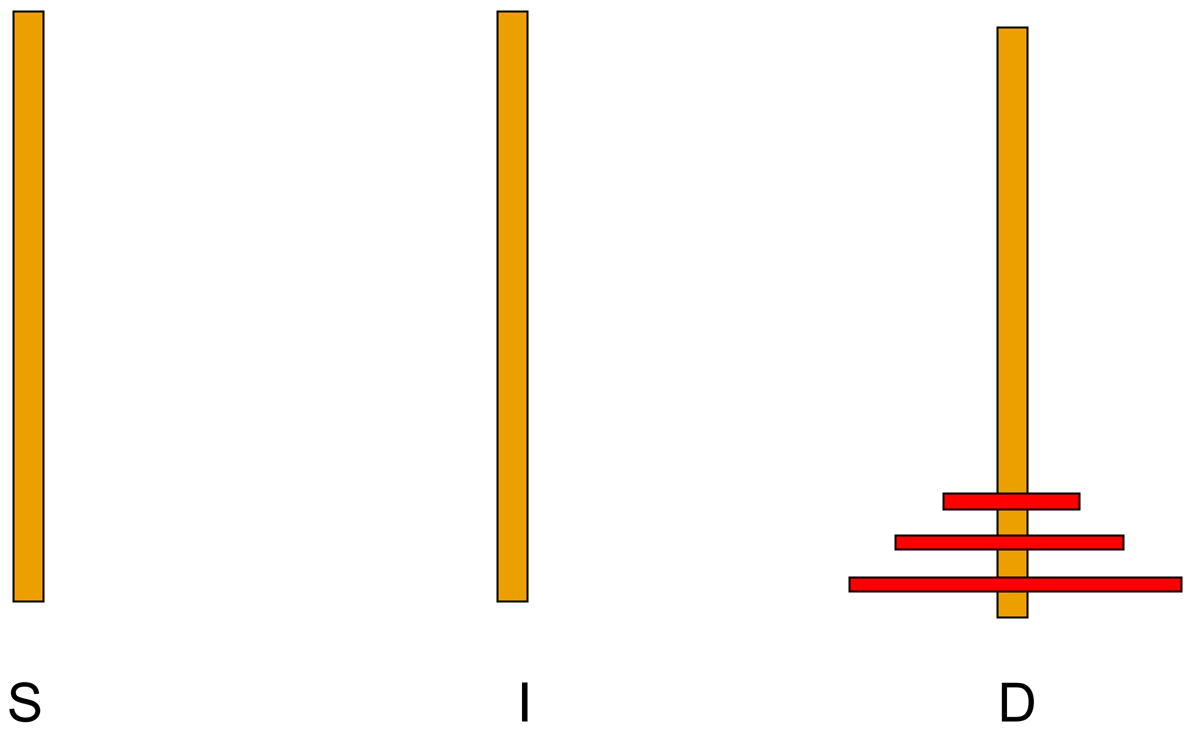
Applications of the reduction technique
The problem of towers in Hanoi

S -> D



Applications of the reduction technique
The problem of towers in Hanoi

S -> D



Applications of the reduction technique The problem of towers in Hanoi

Idea:

Move (n-1) disks from S to I (using D as auxiliary verge)

Move the remaining disk on S directly to D

Move (n-1) disks from I to D (using S as the auxiliary verge)

Algorithm:

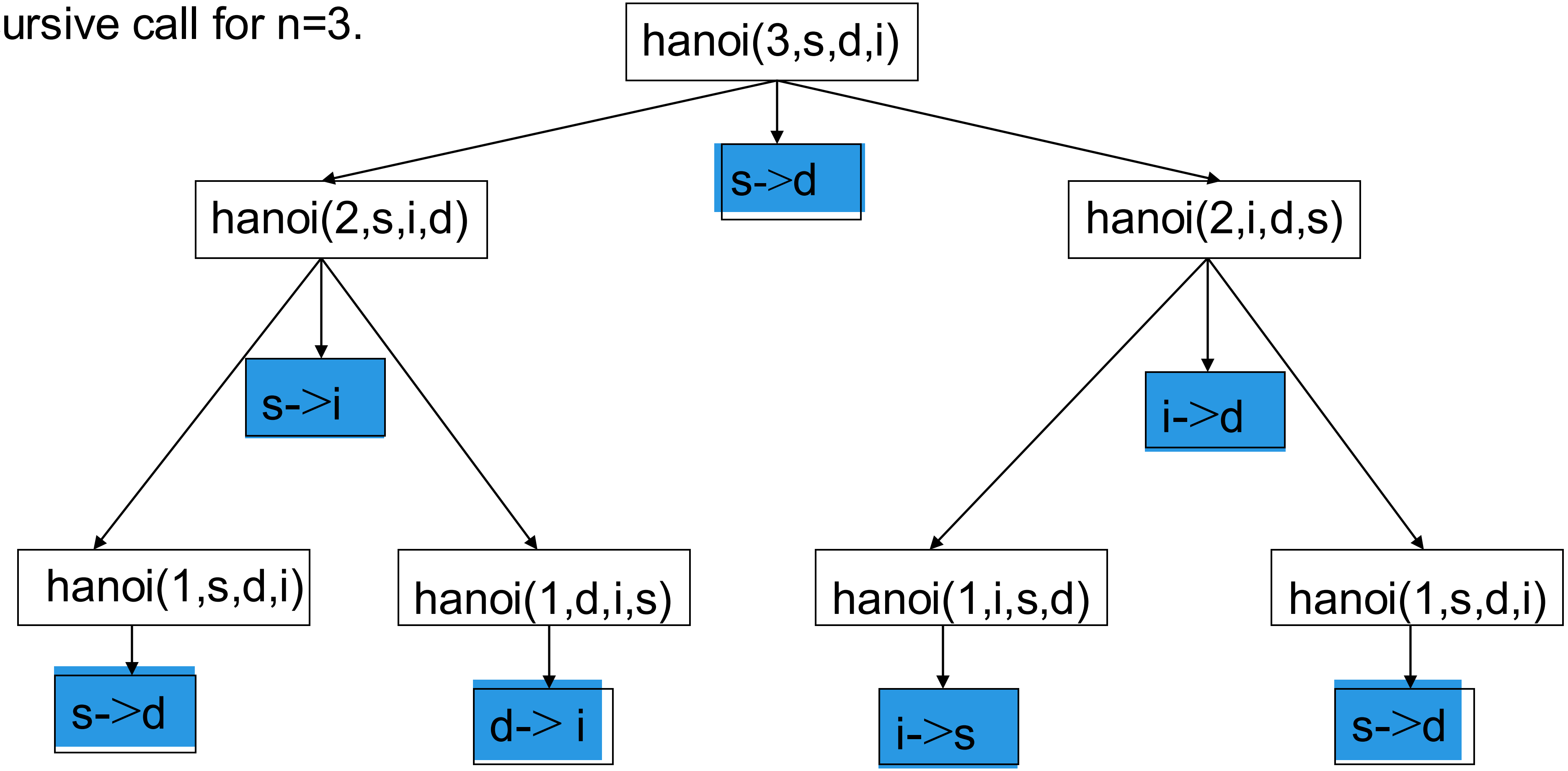
```
hanoi(n,S,D,I)
    IF n=1 THEN   "move from
    S to D"
    ELSE  hanoi(n-1,S,I,D)
          "move from S to D"
          hanoi(n-1,I,D,S)
    ENDIF
```

Meaning of hanoi function parameters:

- First parameter: number of disks
- Second parameter: source verge
- Third parameter: destination verge
- Fourth parameter: intermediate verge

Applications of the reduction technique
The problem of towers in Hanoi

Recursive call for n=3.



Applications of the reduction technique
The problem of towers in 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
```

Dim pb: n
Dominant operation: move
Recurrence relation:

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

$T(n) \quad =2T(n-1)+1$
$T(n-1)=2T(n-2)+1 \quad |*2$
$T(n-2)=2T(n-3)+1 \quad |*2^2$
…
$T(2) \quad =2T(1)+1 \quad |*2^{n-2}$
$T(1) \quad =1 \quad |*2^{n-1}$

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

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

Variants of the reduction technique

- Reduction by subtracting a constant
  - Example: n! (n!=1 if n=1
    n!=(n-1)!*n if n>1)
- Reduction by dividing by a constant
  - Example: x^n ($x^n=x*x$ if n=2
    $x^n=x^{n/2}*x^{n/2}$ if n>2, $n=2^m$)
- Reduction by subtracting a variable value
  - Example: gcd(a,b) (gcd(a,b)=a for a=b
    gcd(a,b)=gcd(b,a-b) for a>b
    gcd(a,b)=gcd(a,b-a) for b>a)
- Reduction by dividing by a variable value
  - Example: gcd(a,b) (gcd(a,b)=a for b=0
    gcd(a,b)=gcd(b,a MOD b) for b<>0)

Next course

… division technique

... analyze

.... applications

Q&A