Course 9:

Algorithms design techniques
- Decrease and conquer -
- Divide and conquer -

Outline

- Brute force
- Decrease-and-conquer
- Recursive algorithms and their analysis
- Applications of decrease-and-conquer
- Divide and conquer

Which are the most used techniques ?

- Brute force
- Decrease and conquer
- Divide and conquer
- Greedy technique
- Dynamic programming
- Backtracking

Brute force

… it is a straightforward approach to solve a problem, usually directly based on the problem's statement

… it is the easiest (and the most intuitive) way for solving a problem

… algorithms designed by brute force are not always efficient

Brute force

Examples:

- Compute x^n, x is a real number and n a natural number

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

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

Complexity

O(n)

Brute force

Examples:

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

Idea: n!=1*2*…*n

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

Complexity

O(n)

Decrease and conquer

Basic idea: exploit the relationship between the solution of a given instance of a problem and the solution of a smaller instance of the same problem. By reducing successively the problem's dimension we eventually arrive to a particular case which can be solved directly.

Motivation:

- such an approach could lead us to an algorithm which is more efficient than a brute force algorithm
- sometimes it is easier to describe the solution of a problem by referring to the solution of a smaller problem than to describe explicitly the solution

Decrease and conquer

Example. Let us consider the problem of computing x^n for $n=2^m$, $m>=1$

Since

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

It follows that we can compute $x^{2^\wedge m}$ by computing:

m=1 => $p:=x*x=x^2$

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

m=3 => $p:=p*p=x^4*x^4=x^8$

….

Decrease and conquer

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

Bottom up approach
(start with the smallest
instance of the problem)

Analysis:

a) Correctness

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

b) Complexity

(i) problem size: m

(ii) dominant operation: *

$T(m) = m$

Remark:

$m=\lg n$

Decrease and conquer

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

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

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

decrease by a constant

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

decrease by a constant factor

Decrease and conquer

```
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
```

Remarks:

1. Top-down approach (start with the largest instance of the problem)
2. Both algorithms are recursive algorithms

Decrease and conquer

This idea can be extended to the case of an arbitrary value for n:

$$
x^n = \begin{cases} x*x & \text{if } n=2 \\ x^{n/2}*x^{n/2} & \text{if } n>2,\ n \text{ is even} \\ x^{(n-1)/2}*x^{(n-1)/2}*x & \text{if } n>2,\ n \text{ is odd} \end{cases}
$$

```
power5(x,n)
    IF  n=1 THEN RETURN x
    ELSE
        IF  n=2 THEN RETURN x*x
        ELSE
            p←power5(x,n DIV 2)
            IF n MOD 2=0 THEN  RETURN p*p
                                ELSE   RETURN p*p*x
    ENDIF ENDIF ENDIF
```

Outline

- Brute force
- Decrease-and-conquer
- Recursive algorithms and their analysis
- Applications of decrease-and-conquer

Recursive algorithms

Definitions

- Recursive algorithm = an algorithm which contains at least one recursive call
- Recursive call = call of the same algorithm either directly (algorithm A calls itself) or indirectly (algorithm A calls algorithm B which calls algorithm A)

Remarks:

- Each recursive algorithm must contain a base case for which it returns the result without calling itself again
- The recursive algorithms are easy to implement but their implementation is not always efficient

Recursive calls - example

fact(4): stack = [4]

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

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

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

fact(4)	24

stack = []

4*fact(3) 4*6

fact(3)	6

stack = [4]

3*fact(2) 3*2

fact(2)	2

stack = [3,4]

2*fact(1) 2*1

fact(1)	1

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

Sequence of recursive calls

Back to the calling function

Recursive algorithms - correctness

Correctness analysis.

to prove that a recursive algorithm is correct it suffices to show that:

- The recurrence relation which describes the relationship between the solution of the problem and the solution for other instances of the problem is correct (from a mathematical point of view)

Correctness can be proved by identifying an assertion (similar to a loop invariant) which has the following properties:

- It is true for the base case
- It remains true after the recursive call
- For the actual values of the algorithm parameters It implies the postcondition

Recursive algorithms-correctness

Example. P: a,b naturals, a<>0; Q: returns gcd(a,b)

Recurrence relation:

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

```
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)

Base case: b=0 => rez=a=gcd(a,b)

After the recursive call: since for b<>0 gcd(a,b)=gcd(b,a MOD b) it follows that rez=gcd(a,b)

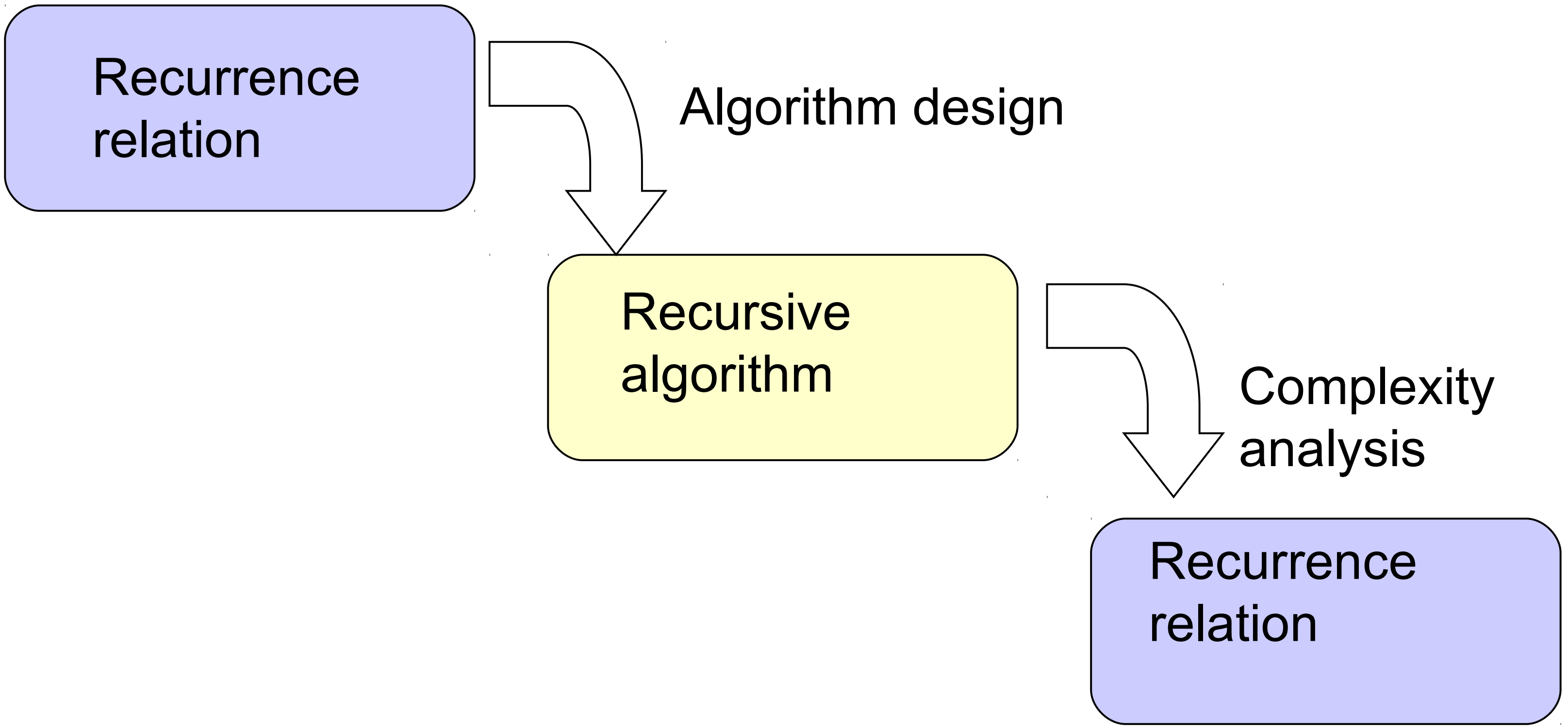
Postcondition: rez=gcd(a,b) => Q

Recursive algorithms - complexity

- Set up a recurrence relation which describes the relation between the running time corresponding to the problem and that corresponding to a smaller instance of the problem. Establish the initial value (based on the particular case). Solve the recurrence relation

Recursive algorithms - efficiency

Remark:



Recursive algorithms - complexity

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

Assumptions:

- <P> is a processing step of constant cost (c0)
- h is a decreasing function and it exists k such that
  $h^{(k)}(n)=h(h(\ldots(h(n))\ldots))=n0$
- The cost of computing h(n) is c

The recurrence relation for the running time is:

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

Recursive algorithms - complexity

Computing n!, $n>=1$

Recurrence relation:

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

Algorithm:

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

Problem dimension: n

Dominant operation: multiplication

Recurrence relation for the running time:

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

Recursive algorithms - complexity

Methods to solve the recurrence relations:

- Forward substitution
  - Start from the base case and construct terms of the sequence
  - Identify a pattern in the sequence and infer the formula of the general term
  - Prove by mathematical induction that the inferred formula satisfies the recurrence relation
- Backward substitution
  - Start from the general case T(n) and replace T(h(n)) with the right-hand side of the corresponding relation, then replace T(h(h(n))) and so on, until we arrive to the particular case
  - Compute the expression of T(n)

Recursive algorithms - complexity

Example: n!

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

Forward substitution

T(1)=0

T(2)=1

T(3)=2

….

T(n)=n-1

It satisfies the recurrence relation

Backward substitution

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

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

….

T(2) =T(1)+1

T(1) =0

------------------------ (by adding)

T(n)=n-1

Remark: same complexity as of the brute force algorithm !

Recursive algorithms - complexity

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
```

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

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

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

….

$T(2) = 1$

------------------------ (by adding)

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

Recursive algorithms - complexity

Remark: for this example decrease and conquer is more efficient than brute force

Explanation: $x^{n/2}$ computed only once. If it would be computed twice then … it is no more decrease and conquer .

```
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 adding)

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

Outline

- Brute force
- Decrease-and-conquer
- Recursive algorithms and their analysis
- Applications of decrease-and-conquer

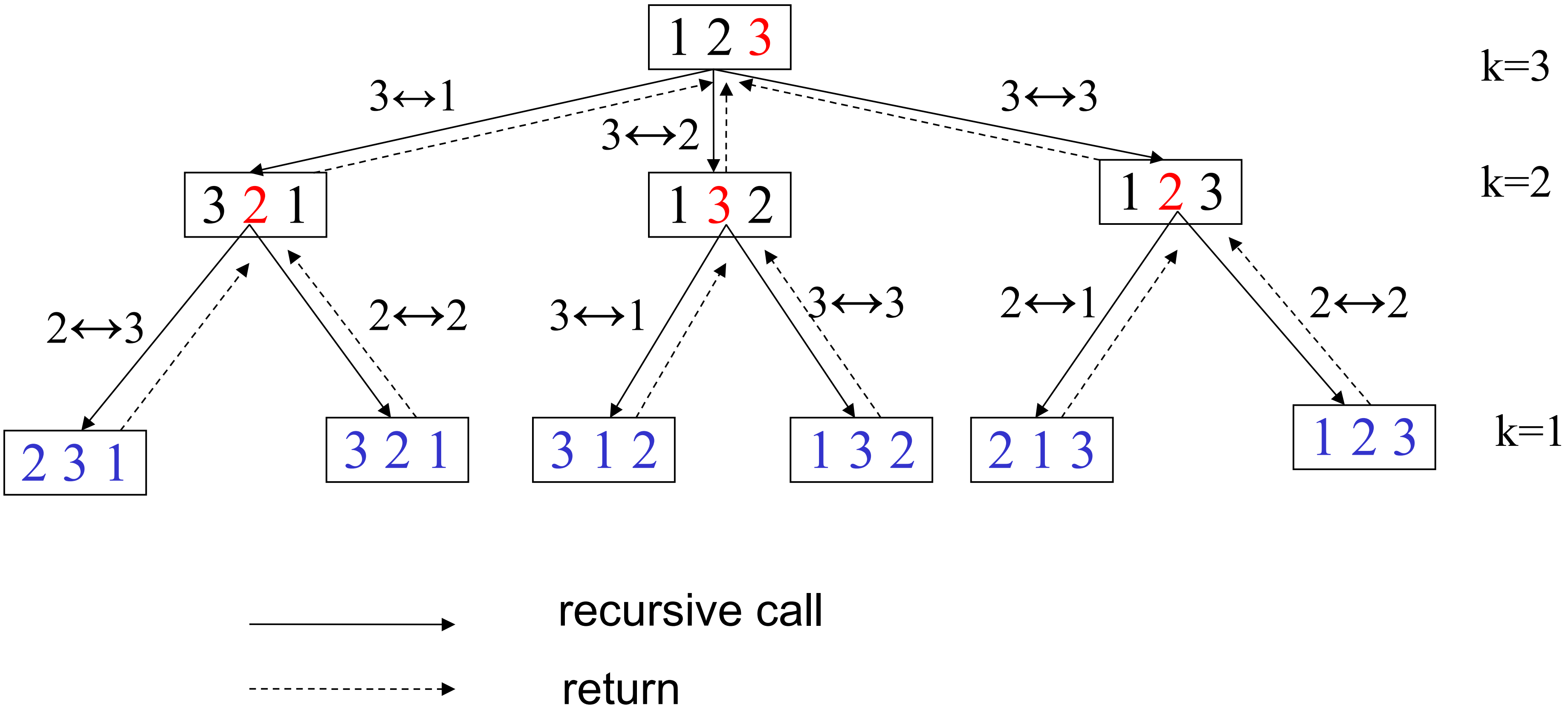
Applications of decrease-and-conquer

Example 1: generating all n! permutations of {1,2,…,n}

Idea: the k! permutations of {1,2,…,k} can be obtained from the (k-1)! permutations of {1,2,…,k-1} by placing the k-th element successively on the first position, second position, third position, … k-th position. Placing k on position i is realized by swapping k with i.

Generating permutations

Illustration for n=3 (top-down approach)



Generating permutations

- Let x[1..n] be a global array (accessed by all functions) containing at the beginning the values (1,2,…,n)
- The algorithm has a formal parameter, k. It is called for k=n.
- The particular case corresponds to k=1, when a permutation is obtained and it is printed.

```
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
```

Remark: the algorithm contains k recursive calls

Complexity analysis:

Problem size: k

Dominant operation: swap

Recurrence relation:

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

Generating permutations

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

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

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

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

…

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

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

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

-> $2e\, k!$ (for large values of k). For k=n => $T(n) \in \Theta(n!)$

Towers of Hanoi

Hypotheses:

- Let us consider three rods labeled S (source), D (destination) and I (intermediary).
- Initially on the rod S there are n disks of different sizes in decreasing order of their size: the largest on the bottom and the smallest on the top

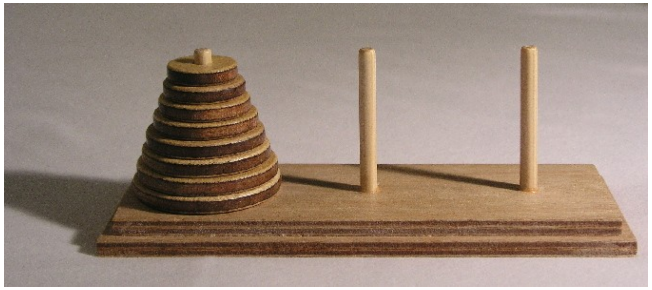
Goal:

- Move all disks from S to D using (if necessary) the rod I as an auxiliary

Restriction:

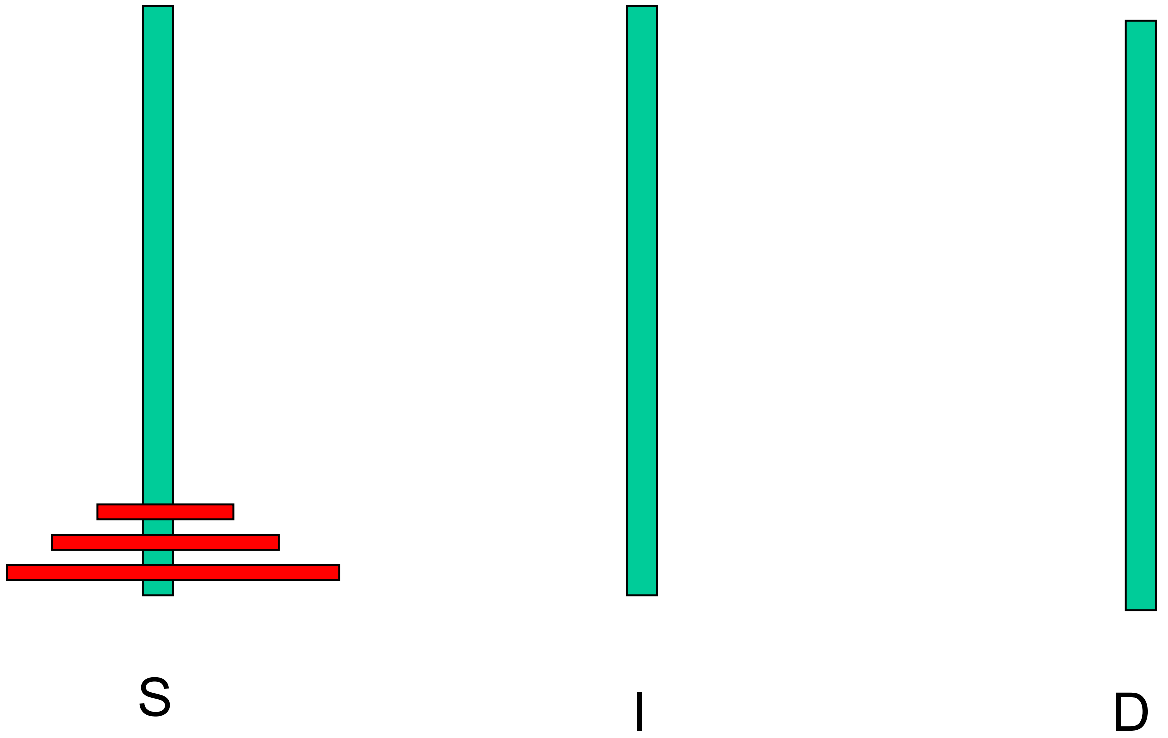
- We can move only one disk at a time

and it is forbidden to place a larger disk on top of a smaller one



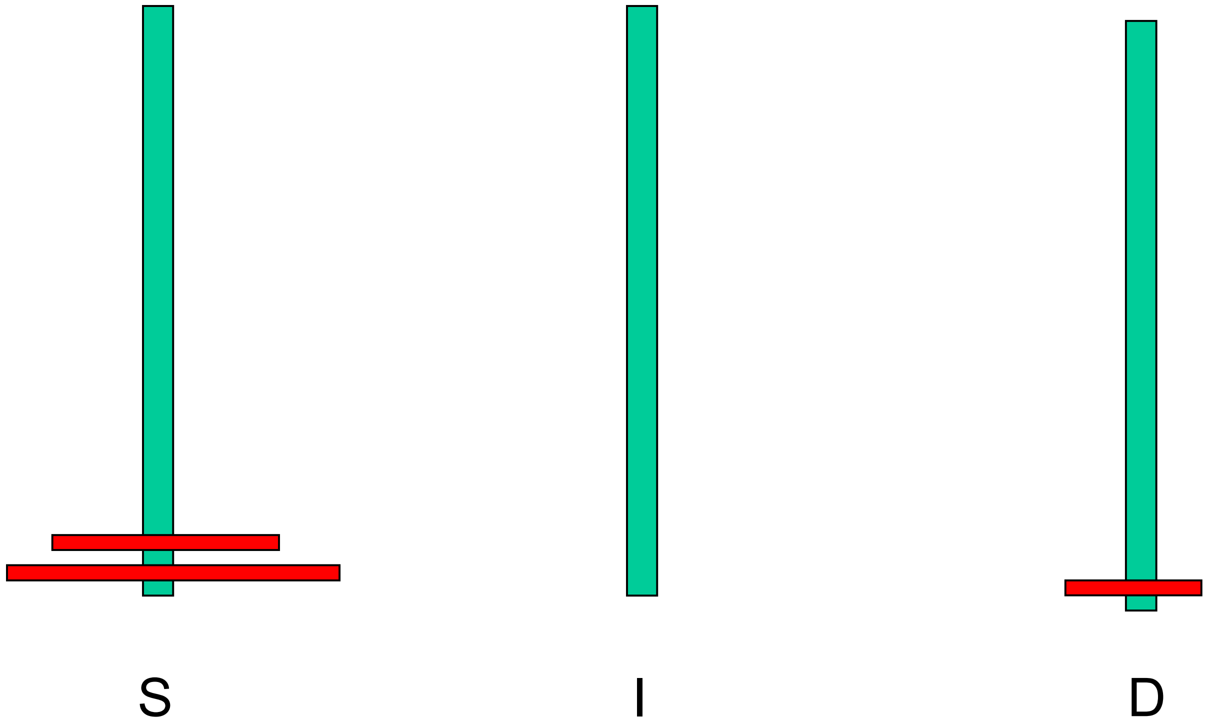
Towers of Hanoi

Initial configuration



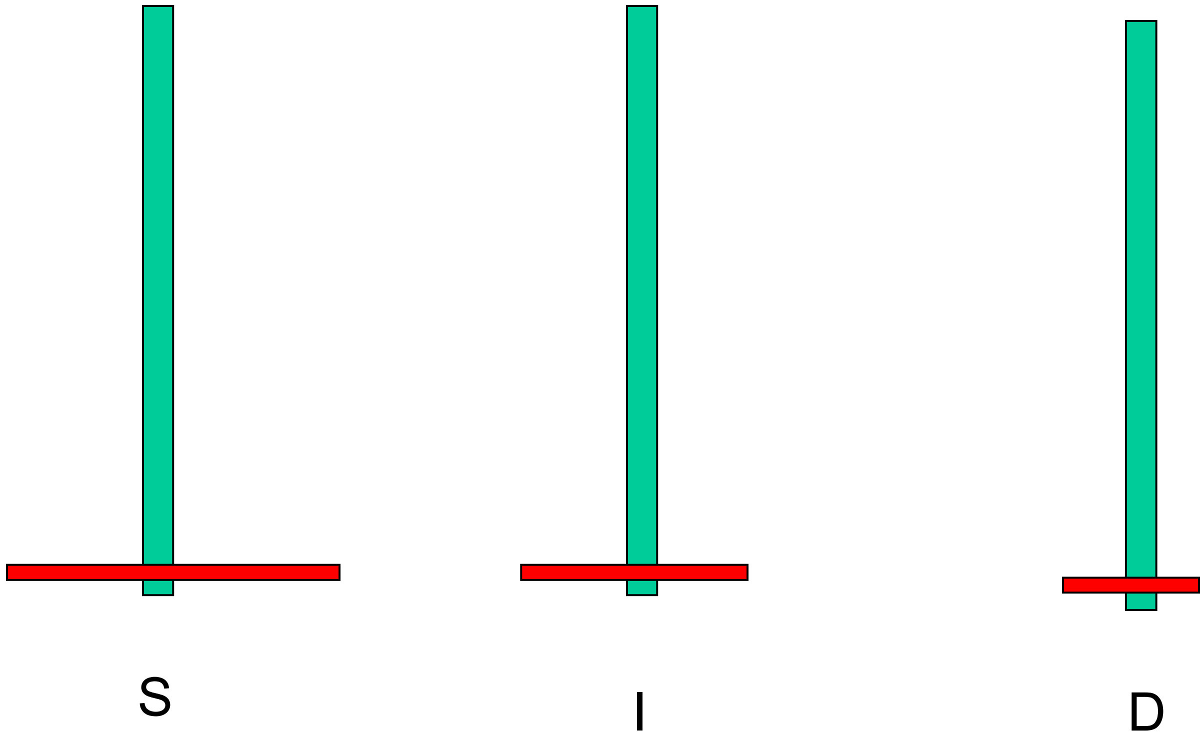
Towers of Hanoi

Move 1:   S->D



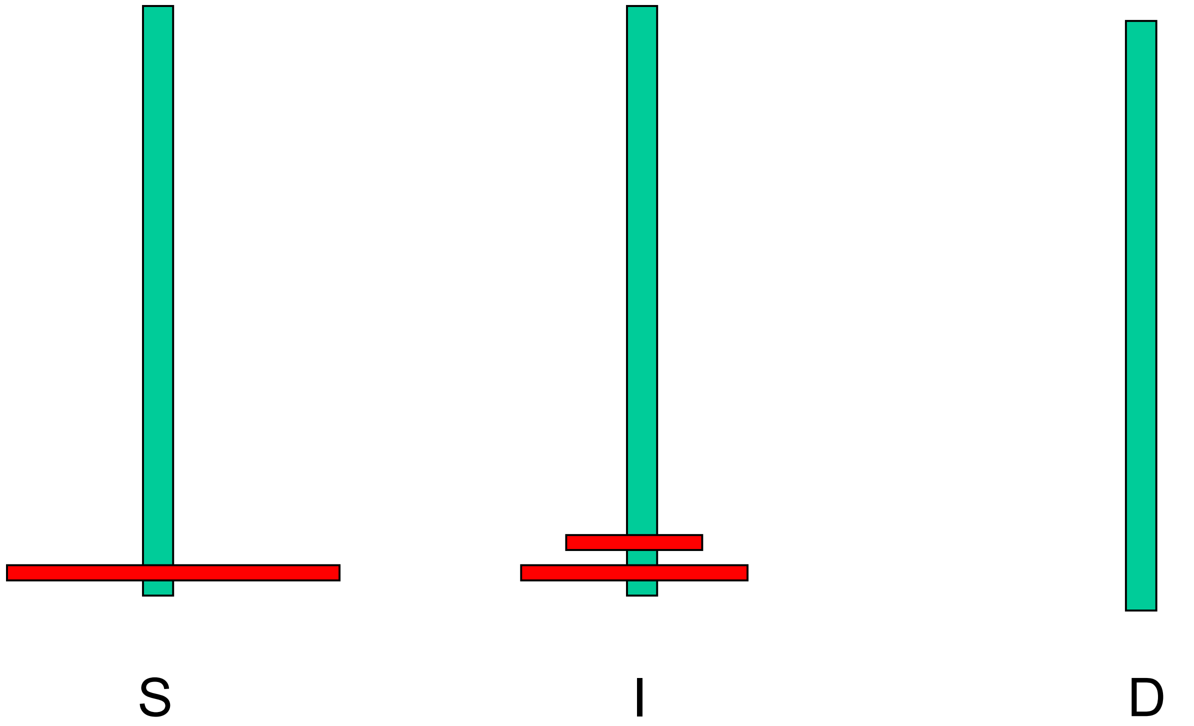
Towers of Hanoi

Move 2:   S->I



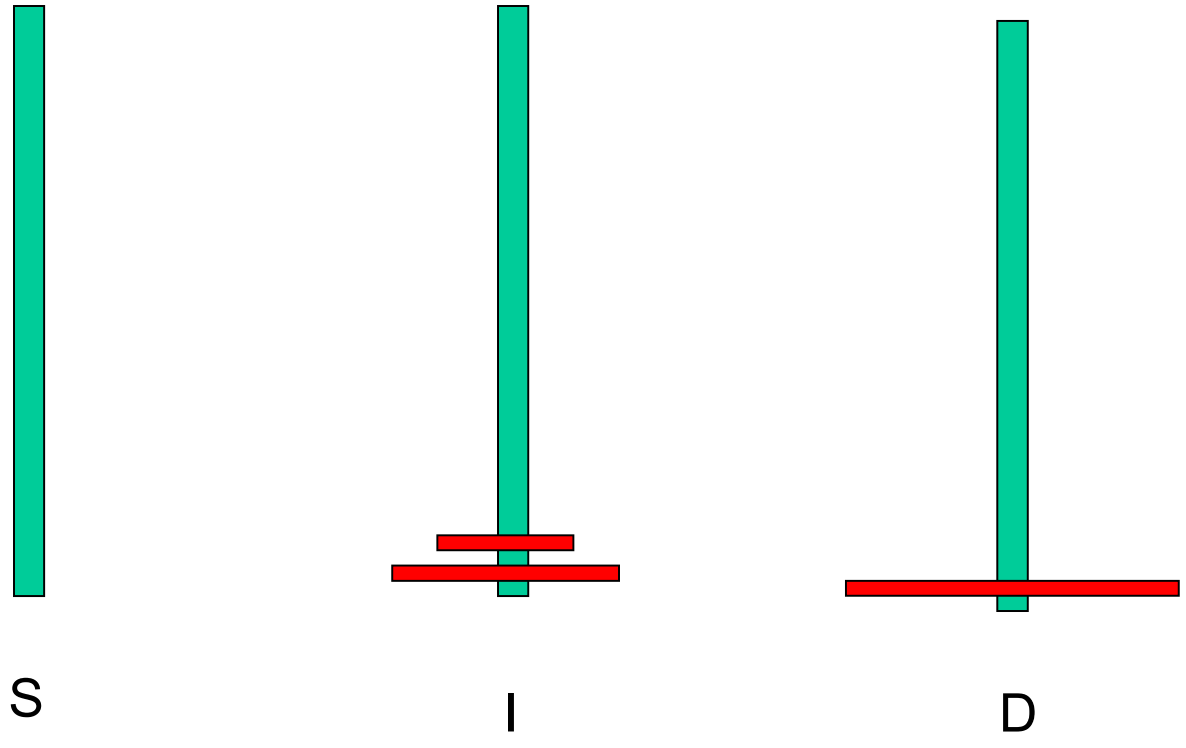
Towers of Hanoi

Move 3: D->I



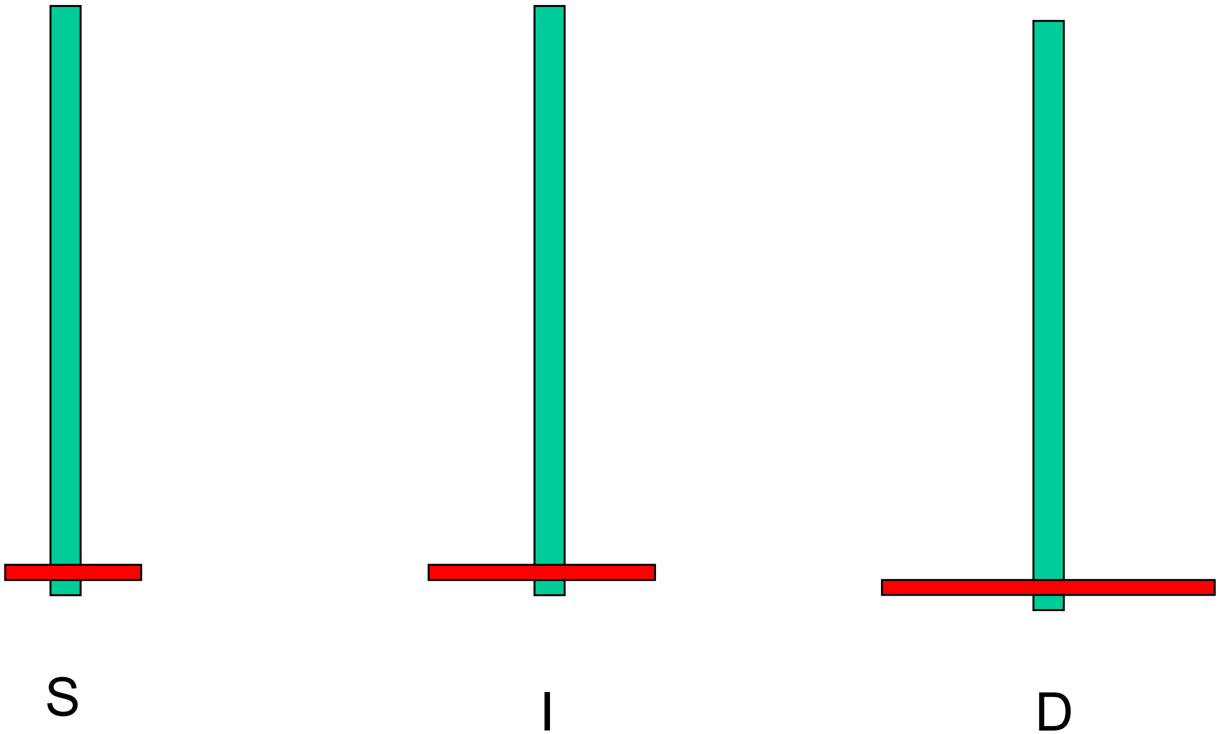
Towers of Hanoi

Move 4:   S->D



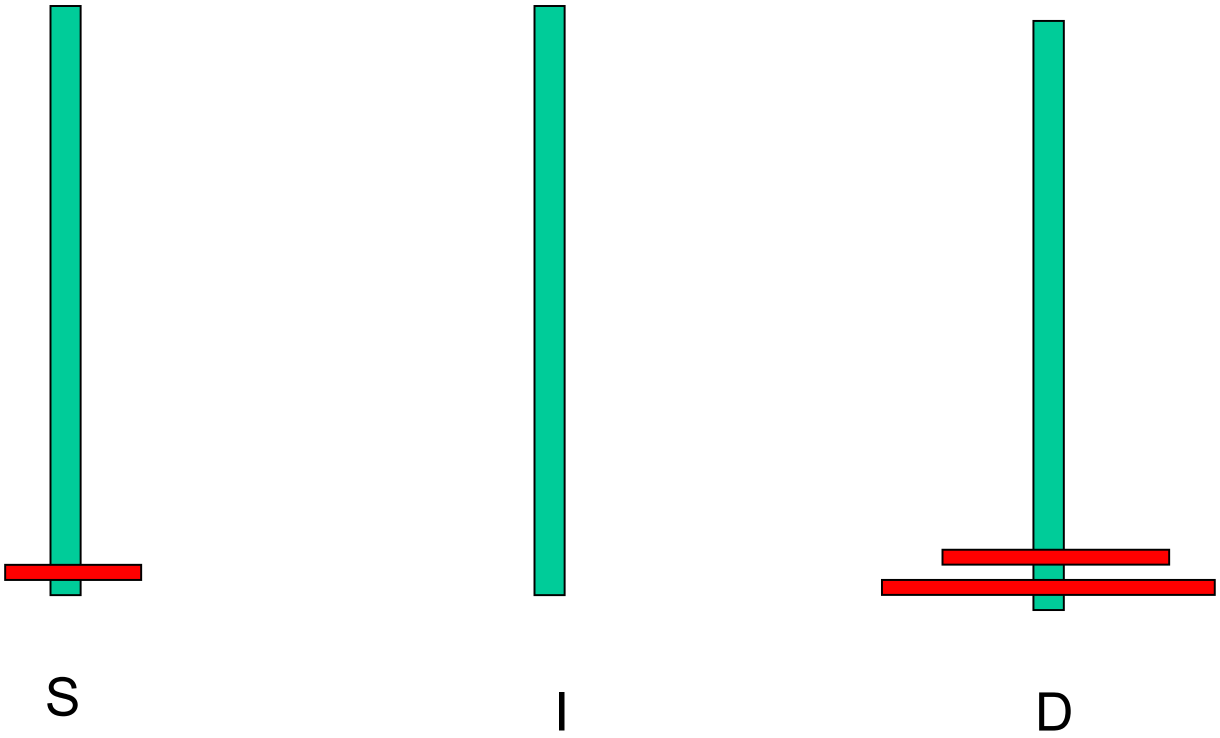
Towers of Hanoi

Move 5: I->S



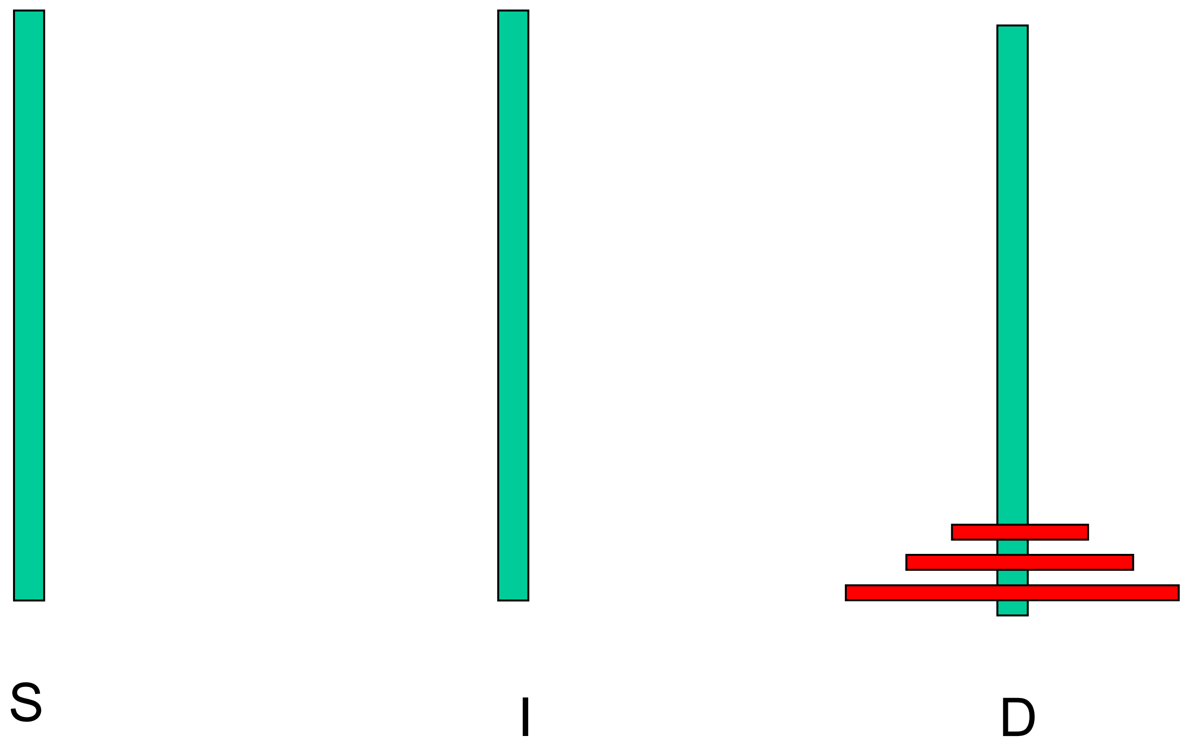
Towers of Hanoi

Move 6:   I->D



Towers of Hanoi

Move 7: S->D



Towers of Hanoi

Idea:

- move (n-1) disks from the rod S to I (using D as auxiliary)
- move the element left on S directly to D
- move the (n-1) disks from the rod I to D (using S as auxiliary)

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
```

Significance of parameters:

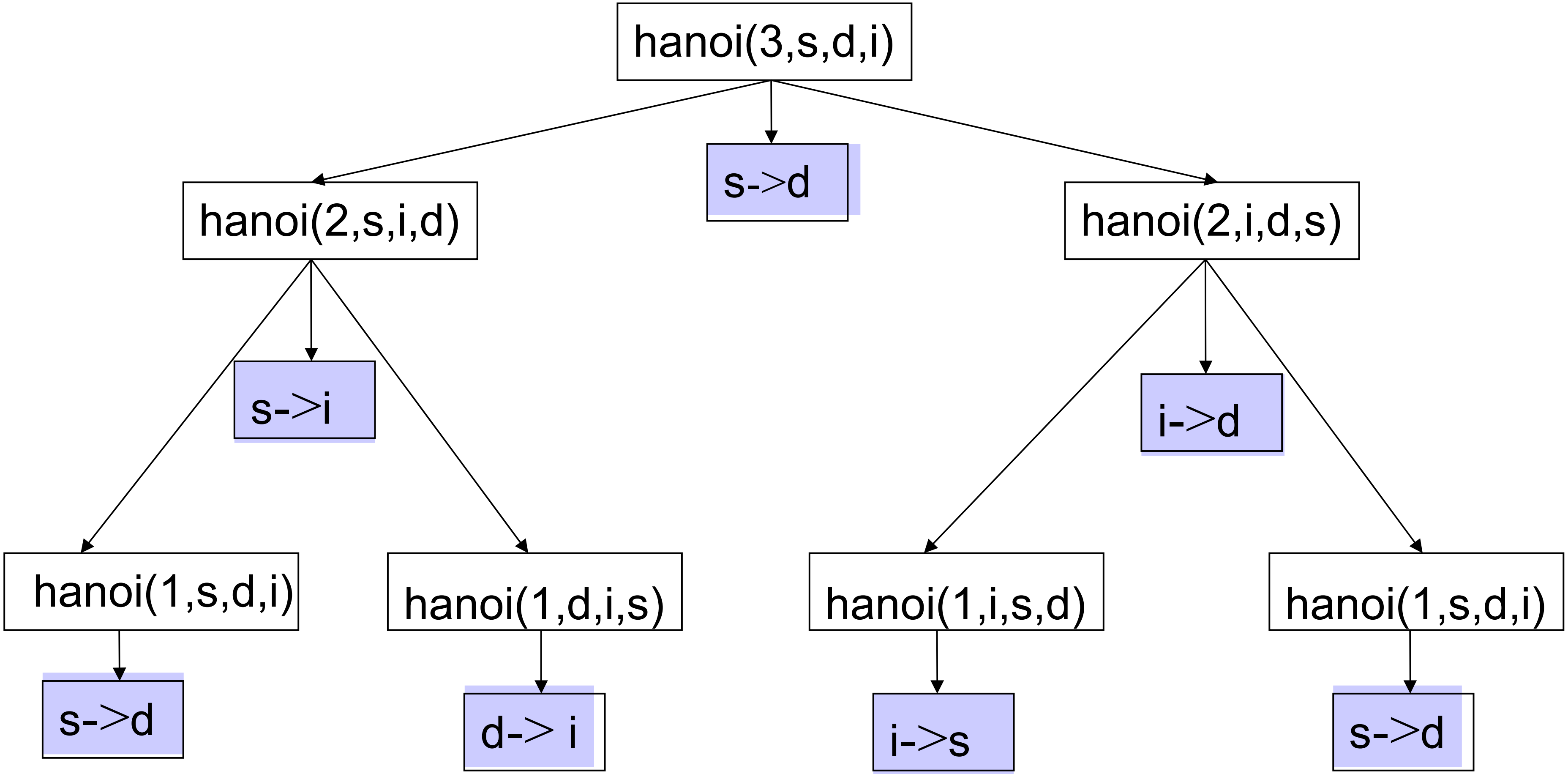
- First parameter: number of disks to be moved
- Second parameter: source rod
- Third parameter: destination rod
- Fourth parameter: auxiliary rod

Remark:

The algorithm contains 2 recursive calls

Towers of Hanoi

Illustration for n=3.



Towers of 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
```

Problem size: 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)□□□2^n)

Basic idea of divide and conquer

- The problem is divided in several smaller instances of the same problem
  - The subproblems must be **independent** (each one will be solved at most once)
  - They should be of about the same size
- These subproblems are solved (by applying the same strategy or directly – if their size is small enough)
  - If the subproblem size is less than a given value (critical size) it is solved directly, otherwise it is solved recursively
- If necessary, the solutions obtained for the subproblems are combined

Basic idea of divide and conquer

Divide&conquer (n)

IF $n \le n_c$ THEN <solve P(n) directly to obtain r>

ELSE

<Divide P(n) in $P(n_1), \ldots, P(n_k)$>

FOR $i \leftarrow 1,k$ DO

$r_i \leftarrow$ Divide&conquer(n_i)

ENDFOR

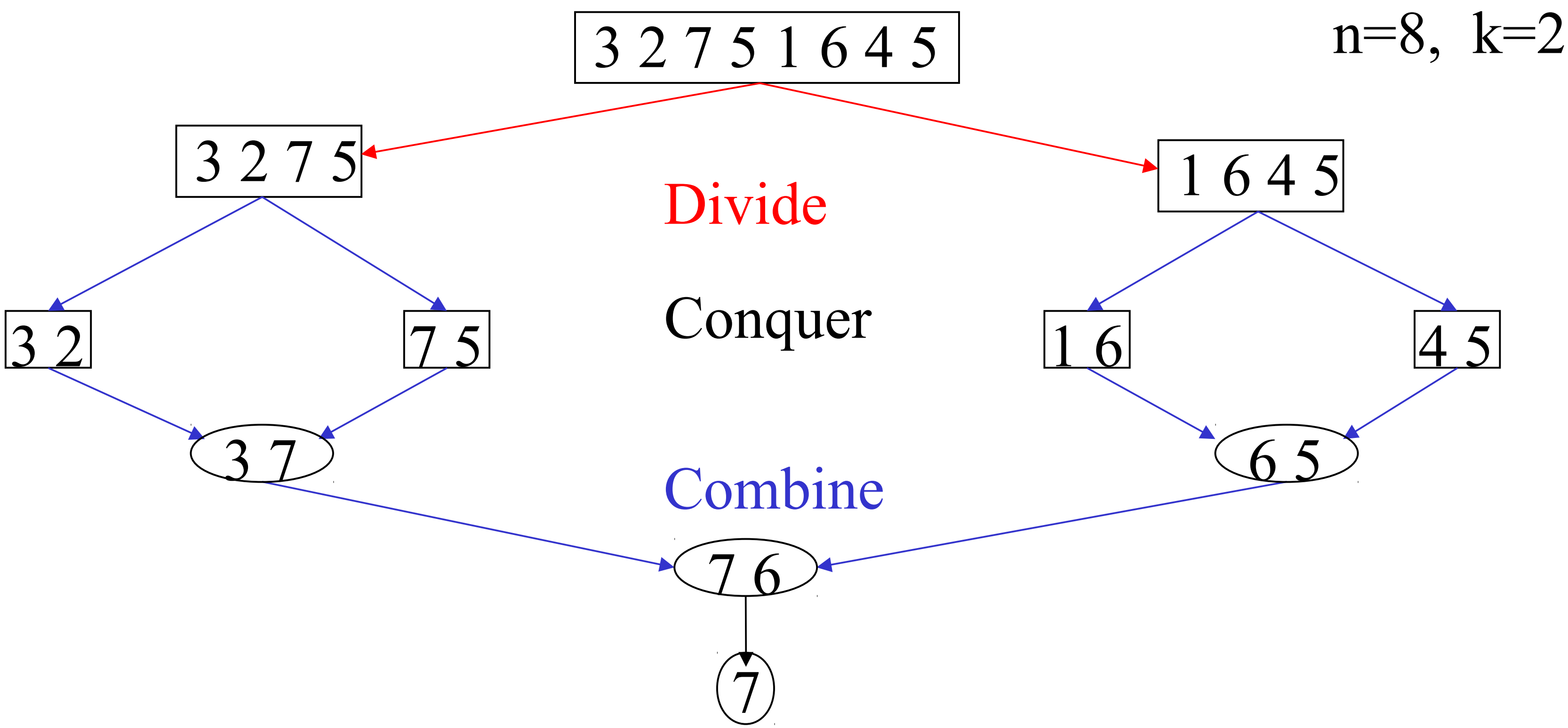
$r \leftarrow$ Combine $(r_1, \ldots r_k)$

ENDIF

RETURN r

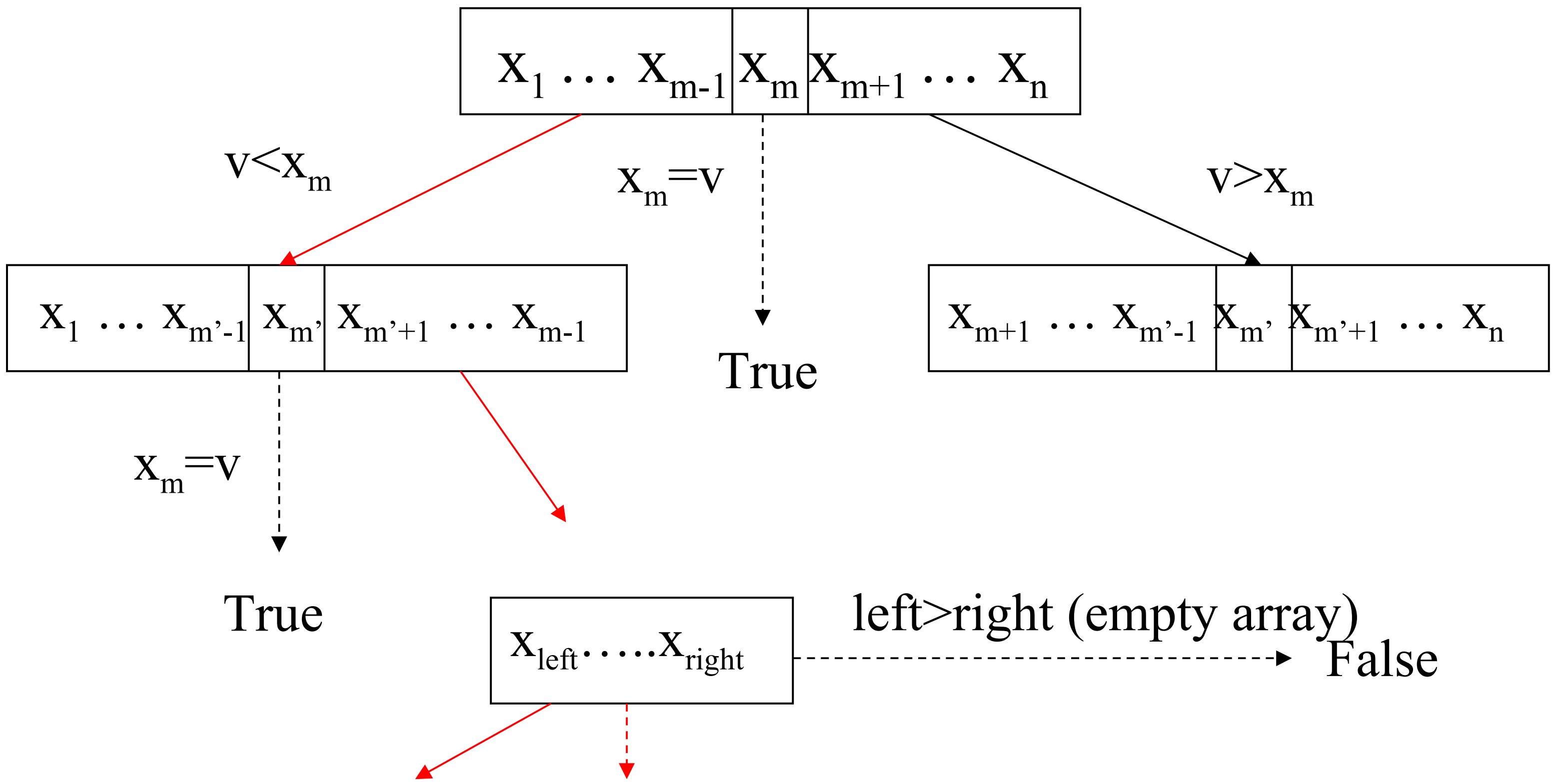
Example 1

Compute the maximum of an array x[1..n]



Example 2 – binary search

Check if a given value, v, is an element of an increasingly sorted array, $x[1..n]$ $(x[i]<=x[i+1])$



Example 2 – binary search

Recursive variant:

```
binsearch(x[left..right],v)
IF left>right THEN RETURN False
ELSE
    m ←(left+right) DIV 2
    IF v=x[m] THEN RETURN True
    ELSE
        IF v<x[m]
         THEN RETURN binsearch(x[left..m-1],v)
          ELSE RETURN binsearch(x[m+1..right],v)
        ENDIF
   ENDIF
ENDIF
```

Remarks:

$n_c=0$

$k=2$

Only one of the two subproblems is solved

This is rather a decrease & conquer approach

Example 2 – binary search

Second iterative variant:

```
binsearch(x[1..n],v)
  left ← 1
  right ← n
  WHILE left<right DO
    m ←(left+right) DIV 2
    IF v<=x[m]
          THEN right ← m
           ELSE left ← m+1
     ENDIF / ENDWHILE
  IF x[left]=v THEN RETURN True
               ELSE RETURN  False
  ENDIF
```

Correctness

Precondition: n>=1

Postcondition:

"returns True if v is in x[1..n] and False otherwise"

Loop invariant: "if v is in x[1..n] then it is in x[left..right]"

(i) left=1, right=n => the loop invariant is true

(ii) It remains true after the execution of the loop body

(iii) when right=left it implies the postcondition

Example 2 – binary search

Second iterative variant:

```
binsearch(x[1..n],v)
  left ← 1
  right ← n
  WHILE left<right DO
     m ←(left+right) DIV 2
     IF v<=x[m]
              THEN right ← m
               ELSE left ← m+1
     ENDIF / ENDWHILE
  IF x[left]=v THEN RETURN True
                   ELSE RETURN  False
  ENDIF
```

Efficiency:

Worst case analysis ($n=2^m$)

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

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

$T(n/2)=T(n/4)+1$

…

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

$T(1)=1$

$T(n)=\lg n+1 \qquad O(\lg n)$

Example 3: mergesort

Basic idea:

- Divide x[1..n] in two subarrays x[1..[n/2]] and x[[n/2]+1..n]
- Sort each subarray
- Merge the elements of x[1..[n/2]] and x[[n/2]+1..n] and construct the sorted temporary array t[1..n] . Transfer the content of the temporary array in x[1..n]

Remarks:

- Base case: 1 (an array containing one element is already sorted)
- Base case can be larger than 1 (e.g.10) and for the particular case one applies a basic sorting algorithm (e.g. insertion sort)