7.MATHEMATICAL ANALYSIS FOR RECURSIVE ALGORITHMS:

General Plan for Analyzing the Time Efficiency of Recursive Algorithms

1. Decide on a *parameter* (or parameters) indicating an input's size.
2. Identify the algorithm's *basic operation*.
3. Check whether the *number of times the basic operation is executed* can vary on different inputs of the same size; if it can, the worst-case, average-case, and best-case *efficiencies* must be investigated separately.
4. *Set up a recurrence relation*, with an appropriate initial condition, for the number of times the basic operation is executed.
5. Solve the recurrence or, at least, ascertain the *order of growth* of its solution.

EXAMPLE 1: Compute the factorial function F(n) = n! for an arbitrary non negative integer n. Since $n! = 1 \cdot \ldots \cdot (n-1) \cdot n = (n-1)! \cdot n$, for $n \ge 1$ and $0! = 1$ by definition, we can compute $F(n) = F(n-1) \cdot n$ with the following recursive algorithm.**(ND 2015) ALGORITHM***F(n)*

```
//Computes n! recursively
//Input: A nonnegative integer n
//Output: The value of n!
if n = 0 return 1
else return F(n − 1) * n
```

Algorithm analysis

- For simplicity, we consider *n* itself as an indicator of this algorithm's input size. i.e.1.
- The basic operation of the algorithm is multiplication; whose number of executions we denote *M(n).* Since the function *F(n)* is computed according to the formula $F(n) = F(n-1) \cdot n$ for $n > 0$.
- The number of multiplications *M(n)* needed to compute it must satisfy the equality

$$M(n) = \underbrace{M(n-1)}_{\text{To compute } F(n-1)} + \underbrace{1}_{\text{To multiply } F(n-1) \text{ by } n} \quad \text{for } n > 0$$

$M(n-1)$ multiplications are spent to compute $F(n-1)$, and one more multiplication is needed to multiply the result by *n*

Recurrence relations

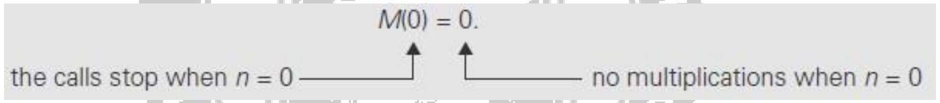
The last equation defines the sequence M(n) that we need to find. This equation defines M(n) not explicitly, i.e., as a function of n, but implicitly as a function of its value at another point, namely n − 1. Such equations are called *recurrence relations* or *recurrences*.

Solve the recurrence relation (n) = (n − 1) + 1, i.e., to find an explicit formula for*M(n)* in terms of *n* only.

To determine a solution uniquely, we need an initial condition that tells us the value with which the sequence starts. We can obtain this value by inspecting the condition that makes the algorithm stop its recursive calls:

if *n* = 0 **return** 1.

This tells us two things. First, since the calls stop when n = 0, the smallest value of n for which this algorithm is executed and hence M(n) defined is 0. Second, by inspecting the pseudocode's exiting line, we can see that when n = 0, the algorithm performs no multiplications.



Thus, the recurrence relation and initial condition for the algorithm's number of multiplications
M(n):
M(n) = M(n − 1) + 1
for n >0, M(0)=0 for
n =0.

Method of backward substitutions

M(n) = *M(n* − 1*)*+1 substitute *M(n* − 1*)* = *M(n* − 2*)* +1
= [*M(n* − 2*)* + 1]+ 1
= *M(n* − 2*)*+2 substitute *M(n* − 2*)* = *M(n* − 3*)* +1
= [*M(n* − 3*)* + 1]+ 2
= *M(n* − 3*)* + 3
…
= M(n − i) + i
…
= M(n − n) + n
= n.

Therefore$M(n)=n$

EXAMPLE 2: consider educational workhorse of recursive algorithms: the***Tower of Hanoi*** puzzle. We have n disks of different sizes that can slide onto any of three pegs. Consider A (source), B (auxiliary), and C (Destination). Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest

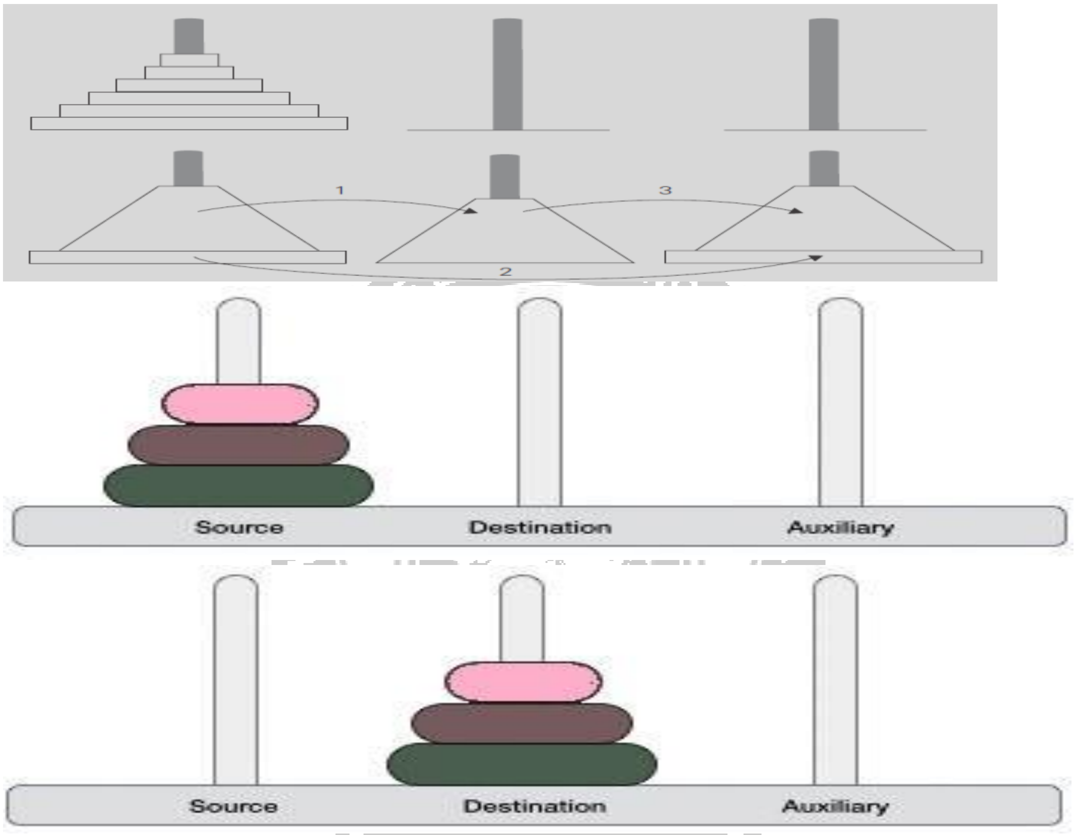


FIGURE 1.7 Recursive solution to the Tower of Hanoi puzzle.

on top. The goal is to move all the disks to the third peg, using the second one as an auxiliary.

```
ALGORITHM TOH(n, A, C, B)
    //Move disks from source to destination recursively
    //Input: n disks and 3 pegs A, B, and C
    //Output: Disks moved to destination as in the source order.
    if n=1
        Move disk from A to C
    else
        Move top n-1 disks from A to B using C
        TOH(n - 1, A, B,C)
        Move top n-1 disks from B to C using A
        TOH(n - 1, B, C, A)
```

Algorithm analysis

The number of moves $M(n)$ depends on n only, and we get the following recurrence equation for it:

$M(n) = M(n-1) + 1 + M(n-1)$ for $n > 1$.

With the obvious initial condition $M(1) = 1$, we have the following recurrence relation for the number of moves $M(n)$:

$$M(n) = 2M(n-1) + 1$$
$$\text{for } n > 1,\ M(1) = 1.$$

We solve this recurrence by the same method of backward substitutions:

$M(n) = 2M(n-1)+1$ sub. $M(n-1) = 2M(n-2) + 1$

$= 2[2M(n-2) + 1] + 1$

$= 2^2M(n-2) + 2 + 1$ sub. $M(n-2) = 2M(n-3) + 1$

$= 2^2[2M(n-3) + 1] + 2 + 1$

$= 2^3M(n-3) + 2^2 + 2 + 1$ sub. $M(n-3) = 2M(n-4) + 1$

$= 2^4M(n-4) + 2^3 + 2^2 + 2 + 1$

…

$= 2^iM(n-i) + 2^{i-1} + 2i^{-2} + \ldots + 2 + 1 = 2^iM(n-i) + 2^i - 1.$

…

Since the initial condition is specified for $n = 1$, which is achieved for $i = n-1$, $M(n) = 2^{n-1}M(n-(n-1)) + 2^{n-1} - 1 = 2^{n-1}M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1$.

Thus, we have an exponential time algorithm

EXAMPLE 3: An investigation of a recursive version of the algorithm which finds the number of binary digits in the **binary representation** of a positive decimal integer.

```
ALGORITHM BinRec(n)
        //Input: A positive decimal integer n
        //Output: The number of binary digits in n's binary representation
        if n = 1 return 1
        else return BinRec(⌊n/2⌋)+1
```

Algorithm analysis

The number of additions made in computing *BinRec*$(\lfloor n/2 \rfloor)$ is $A(\lfloor n/2 \rfloor)$, plus one more addition is made by the algorithm to increase the returned value by 1. This leads to the recurrence $A(n) = A(\lfloor n/2 \rfloor) + 1$ for $n > 1$

Then, the initial condition is $A(1) = 0$.

The standard approach to solving such a recurrence is to solve it

only for $n = 2^k$ $A(2^k) = A(2^{k-1}) + 1$ for $k > 0$,

$A(2^0) = 0$.

backward substitutions

$A(2^k) = A(2^{k-1})+1$ substitute $A(2^{k-1}) = A(2^{k-2}) + 1$

$= [A(2^{k-2}) + 1] + 1 = A(2^{k-2})+2$ substitute $A(2^{k-2}) = A(2^{k-3}) + 1$

$= [A(2^{k-3}) + 1] + 2 = A(2^{k-3})+3$ . . .

. . .

$= A(2^{k-i}) + i$

. . .

$= A(2^{k-k}) + k$.

Thus, we end up with $A(2^k) = A(1) + k = k$, or, after returning to the original variable $n = 2^k$ and hence $k = \log_2 n$,

$A(n) = \log_2 n \in \Theta(\log_2 n)$.