UNIT-2

ASYMPTOTIC NOTATIONS :

Asymptotic Notations:

.Asymptotic Notations are programming languages that allow you to analyze an algorithm's running time by identifying its behavior as its input size grows.
.This is also referred to as an algorithm's growth rate.
.You can't compare two algorithm's head to head.
.You compare space and time complexity using asymptotic analysis.
.It compares two algorithms based on changes in their performance as the input size is increased or decreased.

BASIC EFFICIENCY CLASSES :

Class	Name	Comments
1	*constant*	Short of best-case efficiencies, very few reasonable examples can be given since an algorithm's running time typically goes to infinity when its input size grows infinitely large.
$\log n$	*logarithmic*	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm (see Section 5.5). Note that a logarithmic algorithm cannot take into account all its input (or even a fixed fraction of it): any algorithm that does so will have at least linear running time.
n	*linear*	Algorithms that scan a list of size n (e.g., sequential search) belong to this class.
$n \log n$	"*n-log-n*"	Many divide-and-conquer algorithms (see Chapter 4), including mergesort and quicksort in the average case, fall into this category.
n^2	*quadratic*	Typically, characterizes efficiency of algorithms with two embedded loops (see the next section). Elementary sorting algorithms and certain operations on n-by-n matrices are standard examples.
n^3	*cubic*	Typically, characterizes efficiency of algorithms with three embedded loops (see the next section). Several nontrivial algorithms from linear algebra fall into this class.
2^n	*exponential*	Typical for algorithms that generate all subsets of an n-element set. Often, the term "exponential" is used in a broader sense to include this and larger orders of growth as well.
$n!$	*factorial*	Typical for algorithms that generate all permutations of an n-element set.

ASYMPTOTIC NOTATIONS

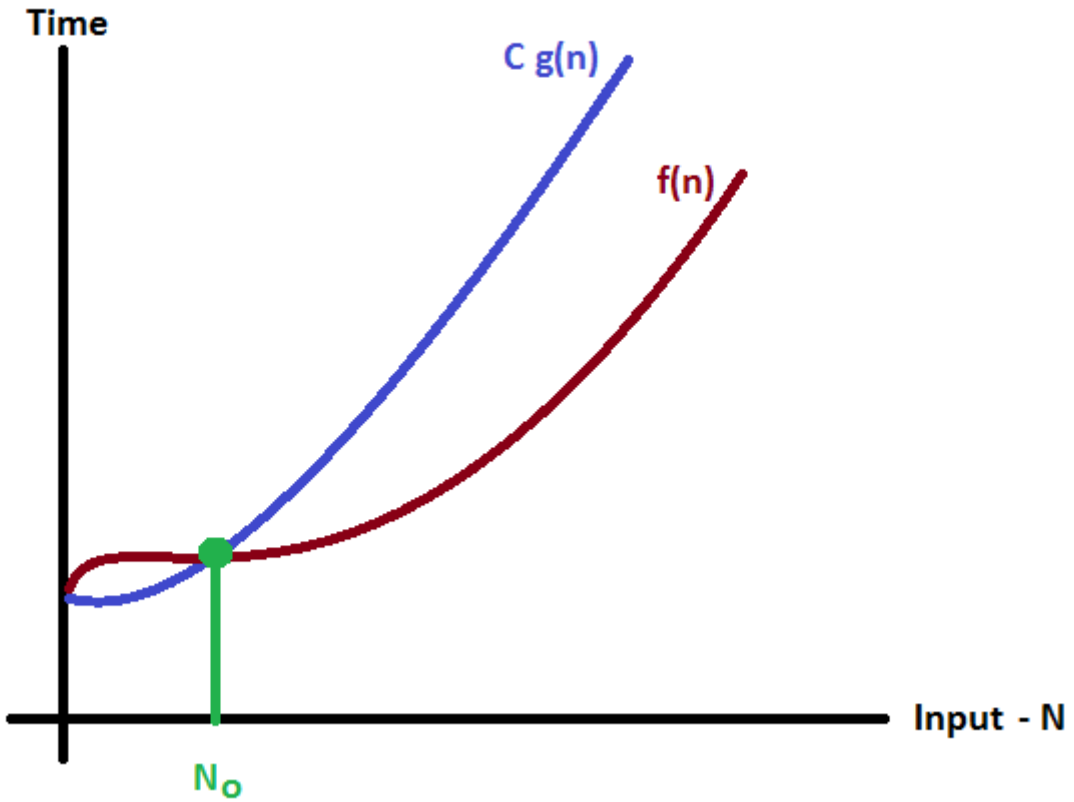
Big - Oh Notation (O)

Big - Oh notation is used to define the **upper bound** of an algorithm in terms of Time Complexity.
That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values. That means Big - Oh notation describes the worst case of an algorithm time complexity.
Big - Oh Notation can be defined as follows...

Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If f(n) <= C g(n) for all n >= n_0, C > 0 and n_0 >= 1. Then we can represent f(n) as O(g(n)).

f(n) = O(g(n))

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C g(n) is greater than f(n) which indicates the algorithm's upper bound.

Example

Consider the following f(n) and g(n)...
f(n) = 3n + 2
g(n) = n
If we want to represent **f(n)** as **O(g(n))** then it must satisfy **f(n) <= C g(n)** for all values of **C > 0** and **n_0>= 1**
f(n) <= C g(n)
⇒3n + 2 <= C n
Above condition is always TRUE for all values of **C = 4** and **n >= 2**.
By using Big - Oh notation we can represent the time complexity as follows...
3n + 2 = O(n)

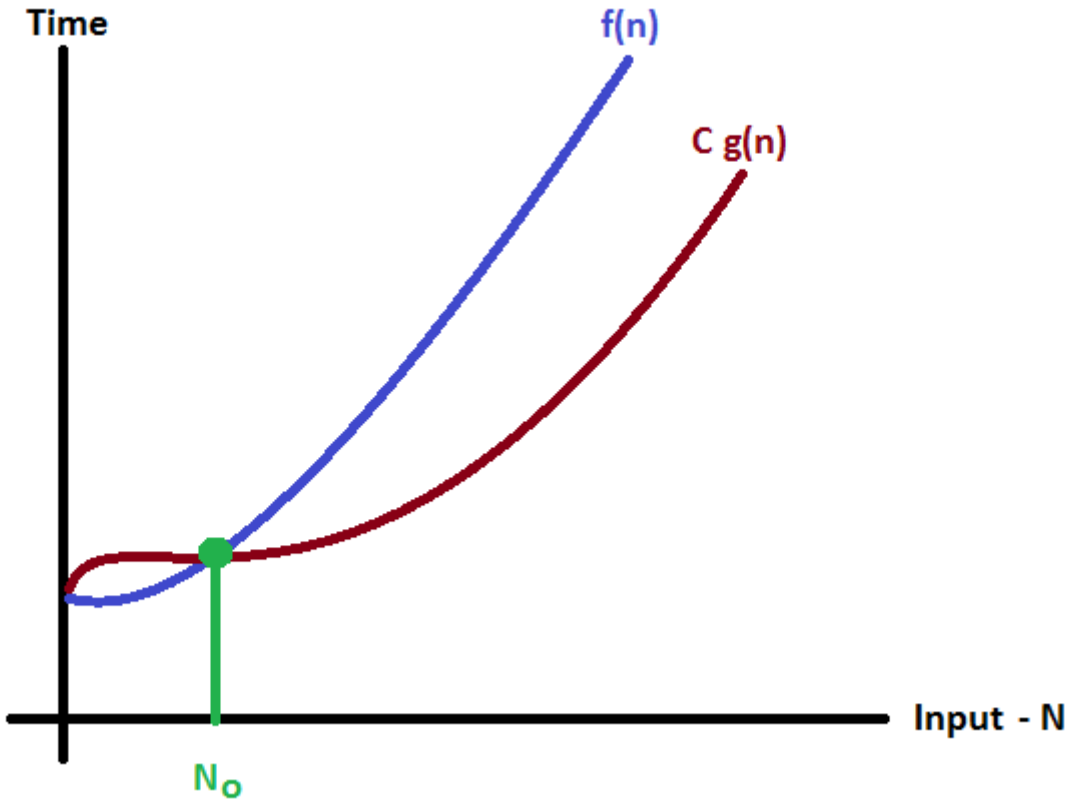
Big - Omege Notation (Ω)

Big - Omega notation is used to define the **lower bound** of an algorithm in terms of Time Complexity.
That means Big-Omega notation always indicates the minimum time required by an algorithm for all input values. That means Big-Omega notation describes the best case of an algorithm time complexity.
Big - Omega Notation can be defined as follows...

Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If f(n) >= C g(n) for all n >= n_0, C > 0 and n_0 >= 1. Then we can represent f(n) as Ω(g(n)).

f(n) = Ω(g(n))

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C g(n) is less than f(n) which indicates the algorithm's lower bound.

Example

Consider the following f(n) and g(n)...
f(n) = 3n + 2
g(n) = n
If we want to represent **f(n)** as **Ω(g(n))** then it must satisfy **f(n) >= C g(n)** for all values of **C > 0** and **n_0>= 1**
f(n) >= C g(n)
⇒3n + 2 >= C n
Above condition is always TRUE for all values of **C = 1** and **n >= 1**.
By using Big - Omega notation we can represent the time complexity as follows...
3n + 2 = Ω(n)

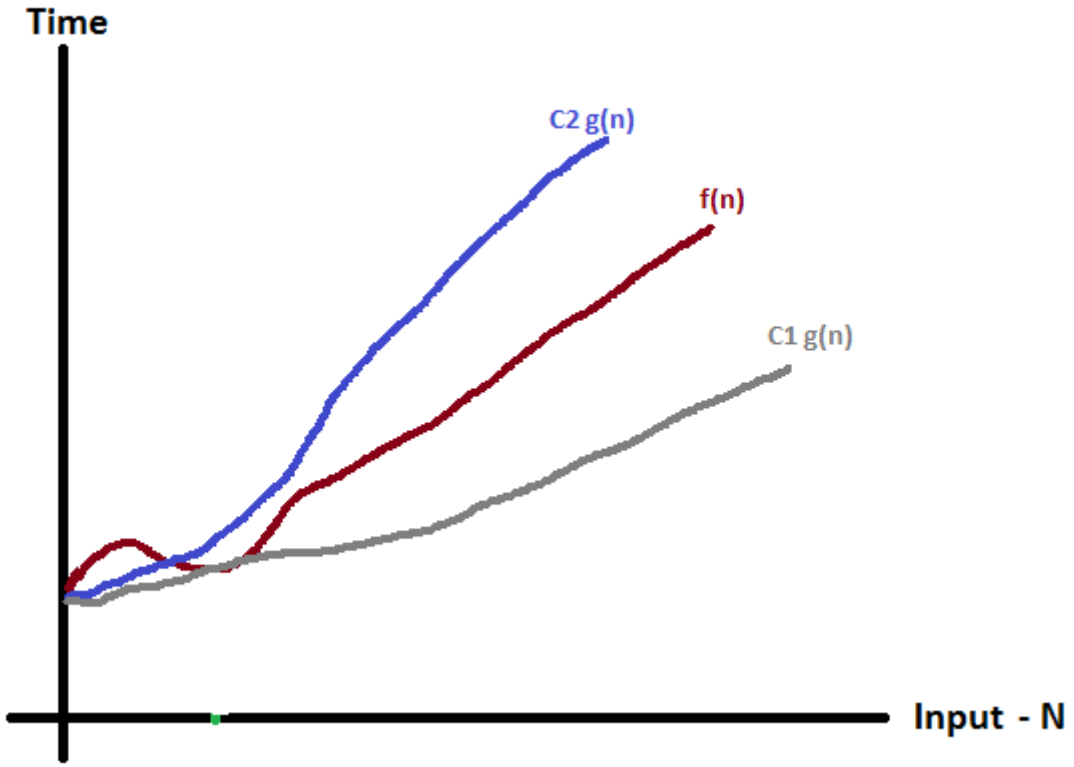
Big - Theta Notation (Θ)

Big - Theta notation is used to define the **average bound** of an algorithm in terms of Time Complexity.
That means Big - Theta notation always indicates the average time required by an algorithm for all input values. That means Big - Theta notation describes the average case of an algorithm time complexity.
Big - Theta Notation can be defined as follows...

Consider function f(n) as time complexity of an algorithm and g(n) is the most significant term. If C_1 g(n) <= f(n) <= C_2 g(n) for all n >= n_0, C_1 > 0, C_2 > 0 and n_0 >= 1. Then we can represent f(n) as Θ(g(n)).

f(n) = Θ(g(n))

Consider the following graph drawn for the values of f(n) and C g(n) for input (n) value on X-Axis and time required is on Y-Axis



In above graph after a particular input value n_0, always C_1 g(n) is less than f(n) and C_2 g(n) is greater than f(n) which indicates the algorithm's average bound.

Example

Consider the following f(n) and g(n)...
f(n) = 3n + 2
g(n) = n
If we want to represent **f(n)** as **Θ(g(n))** then it must satisfy **C_1 g(n) <= f(n) <= C_2 g(n)** for all values of **C_1 > 0, C_2 > 0** and **n_0>= 1**
C_1 g(n) <= f(n) <= C_2 g(n)
⇒C_1 n <= 3n + 2 <= C_2 n
Above condition is always TRUE for all values of **C_1 = 1, C_2 = 4** and **n >= 2**.
By using Big - Theta notation we can represent the time compexity as follows...
3n + 2 = Θ(n)

MATHEMATICAL ANALYSIS OF NON RECURSIVE ALGORITHMS

In this section, we systematically apply the general framework outlined in Section 2.1 to analyzing the time efficiency of nonrecursive algorithms. Let us start with a very simple example that demonstrates all the principal steps typically taken in analyzing such algorithms.

EXAMPLE 1 Consider the problem of finding the value of the largest element in a list of n numbers. For simplicity, we assume that the list is implemented as an array. The following is pseudocode of a standard algorithm for solving the problem.

ALGORITHM *MaxElement*$(A[0..n-1])$

//Determines the value of the largest element in a given array

//Input: An array $A[0..n-1]$ of real numbers

//Output: The value of the largest element in A $maxval \leftarrow A[0]$

for $i \leftarrow 1$ **to** $n-1$ **do**

if $A[i] > maxval$ $maxval \leftarrow A[i]$

return *maxval*

The obvious measure of an input's size here is the number of elements in the array, i.e., n. The operations that are going to be executed most often are in the algorithm's **for** loop. There are two operations in the loop's body: the comparison $A[i] > maxval$ and the assignment $maxval \leftarrow A[i]$. Which of these two operations should we consider basic? Since the comparison is executed on each repetition of the loop and the assignment is not, we should consider the comparison to be the algorithm's basic operation. Note that the number of comparisons will be the same for all arrays of size n; therefore, in terms of this

metric, there is no need to distinguish among the worst, average, and best cases here.

Let us denote ***C(n)*** the number of times this comparison is executed and try to find a formula expressing it as a function of size ***n***. The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable ***i*** within the bounds 1 and ***n*** − 1, inclusive. Therefore, we get the following sum for ***C(n)***:

$$C(n) = \sum_{i=1}^{n-1} 1.$$

This is an easy sum to compute because it is nothing other than 1 repeated ***n*** − 1 times. Thus,

$$C(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

Here is a general plan to follow in analyzing nonrecursive algorithms.

General Plan for Analyzing the Time Efficiency of Nonrecursive Algorithms

Decide on a parameter (or parameters) indicating an input's size.

Identify the algorithm's basic operation. (As a rule, it is located in the inner-most loop.)

Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.

Set up a sum expressing the number of times the algorithm's basic operation is executed.

Using standard formulas and rules of sum manipulation, either find a closed-form formula for the count or, at the very least, establish its order of growth.

Before proceeding with further examples, you may want to review Appen-dix A, which contains a list of summation formulas and rules that are often useful in analysis of algorithms. In particular, we use especially frequently two basic rules of sum manipulation

$$\sum_{i=l}^{u} ca_i = c\sum_{i=l}^{u} a_i, \qquad \textbf{(R1)}$$

$$\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i, \qquad \textbf{(R2)}$$

and two summation formulas

$$\sum_{i=l}^{u} 1 = u - l + 1 \quad \text{where } l \le u \text{ are some lower and upper integer limits,} \qquad \textbf{(S1)}$$

$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2}n^2 \in \Theta(n^2). \qquad \textbf{(S2)}$$

Note that the formula $\sum_{i=1}^{n-1} 1 = n - 1$, which we used in Example 1, is a special case of formula (S1) for $l = 1$ and $u = n - 1$.

EXAMPLE 2 Consider the ***element uniqueness problem***: check whether all the elements in a given array of ***n*** elements are distinct. This problem can be solved by the following straightforward algorithm.

ALGORITHM *UniqueElements*(***A***[0**..*n* − 1])

//Determines whether all the elements in a given array are distinct //Input: An array ***A***[0**..*n* − 1]

//Output: Returns “true” if all the elements in ***A*** are distinct // and “false” otherwise

for $i \leftarrow 0$ **to** $n - 2$ **do**

for $j \leftarrow i + 1$ **to** $n - 1$ **do**

if $A[i] = A[j]$ **return false return true**

The natural measure of the input’s size here is again ***n,*** the number of elements in the array. Since the innermost loop contains a single operation (the comparison of two elements), we should consider it as the algorithm’s basic operation. Note, however, that the number of element comparisons depends not only on ***n*** but also on whether there are equal elements in the array and, if there are, which array positions they occupy. We will limit our investigation to the worst case only.

By definition, the worst case input is an array for which the number of element comparisons $C_{worst}(n)$ is the largest among all arrays of size ***n.*** An inspection of the innermost loop reveals that there are two kinds of worst-case inputs—inputs for which the algorithm does not exit the loop prematurely: arrays with no equal elements and arrays in which the last two elements are the only pair of equal elements. For such inputs, one comparison is made for each repetition of the innermost loop, i.e., for each value of the loop variable ***j*** between its limits $i + 1$ and $n - 1$; this is repeated for each value of the outer loop, i.e., for each value of the loop variable ***i*** between its limits 0 and $n - 2$**.** Accordingly, we get

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$
$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$
$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2).$$

We also could have computed the sum $\sum_{i=0}^{n-2}(n-1-i)$ faster as follows:

$$\sum_{i=0}^{n-2} (n-1-i) = (n-1) + (n-2) + \cdots + 1 = \frac{(n-1)n}{2},$$

where the last equality is obtained by applying summation formula (S2). Note that this result was perfectly predictable: in the worst case, the algorithm needs to compare all ***n(n*** − 1***)/***2 distinct pairs of its ***n*** elements

MATHEMATICAL ANALYSIS OF RECURSIVE ALGORITHMS

In this section, we will see how to apply the general framework for analysis of algorithms to recursive algorithms. We start with an example often used to introduce novices to the idea of a recursive algorithm.

EXAMPLE 1 Compute the factorial function ***F (n)*** = ***n***! for an arbitrary nonnegative integer ***n.*** Since

n! = 1 · **. . .** · ***(n*** − 1***)*** · ***n*** = ***(n*** − 1***)***! · ***n*** for ***n*** ≥ 1

and 0! = 1 by definition, we can compute ***F (n)*** = ***F (n*** − 1***)*** · ***n*** with the following recursive algorithm.

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***

For simplicity, we consider ***n*** itself as an indicator of this algorithm's input size (rather than the number of bits in its binary expansion). The basic operation of the algorithm is multiplication,[5] whose number of executions we denote ***M(n).*** Since the function ***F (n)*** is computed according to the formula

F (n) = ***F (n*** − 1***)*** · ***n*** for ***n*** > 0**,**

the number of multiplications ***M(n)*** needed to compute it must satisfy the equality

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

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

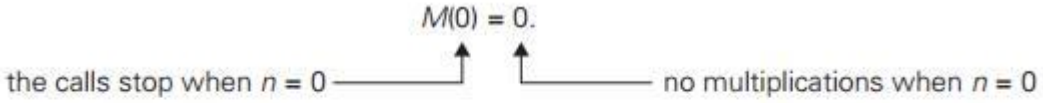
The last equation defines the sequence ***M(n)*** that we need to find. This equa-tion defines ***M(n)*** not explicitly, i.e., as a function of ***n,*** but implicitly as a function of its value at another point, namely ***n*** − 1**.** Such equations are called ***recurrence relations*** or, for brevity, ***recurrences***. Recurrence relations play an important role not only in analysis of algorithms but also in some areas of applied mathematics. They are usually studied in detail in courses on discrete mathematics or discrete structures; a very brief tutorial on them is provided in Appendix B. Our goal now is to solve the recurrence relation ***M(n)*** = ***M(n*** − 1***)*** + 1, i.e., to find an explicit formula for ***M(n)*** in terms of ***n*** only**.**

Note, however, that there is not one but infinitely many sequences that satisfy this recurrence. (Can you give examples of, say, two of them?) 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. Therefore, the initial condition we are after is



Thus, we succeeded in setting up the recurrence relation and initial condition for the algorithm's number of multiplications $M(n)$:

$$M(n) = M(n-1) + 1 \quad \text{for } n > 0, \qquad \textbf{(2.2)}$$
$$M(0) = 0.$$

Before we embark on a discussion of how to solve this recurrence, let us pause to reiterate an important point. We are dealing here with two recursively defined functions. The first is the factorial function ***F (n)*** itself; it is defined by the recurrence

$$F(n) = F(n-1) \cdot n \quad \text{for every } n > 0,$$
$$F(0) = 1.$$

The second is the number of multiplications ***M(n)*** needed to compute ***F (n)*** by the recursive algorithm whose pseudocode was given at the beginning of the section.

As we just showed, ***M(n)*** is defined by recurrence (2.2). And it is recurrence (2.2) that we need to solve now.

Though it is not difficult to "guess" the solution here (what sequence starts with 0 when ***n*** = 0 and increases by 1 on each step?), it will be more useful to arrive at it in a systematic fashion. From the several techniques available for solving recurrence relations, we use what can be called the ***method of backward substitutions***. The method's idea (and the reason for the name) is immediately clear from the way it applies to solving our particular recurrence:

$$
\begin{aligned}
M(n) &= M(n-1) + 1 && \text{substitute } M(n-1) = M(n-2) + 1\\
&= [M(n-2) + 1] + 1 = M(n-2) + 2 && \text{substitute } M(n-2) = M(n-3) + 1\\
&= [M(n-3) + 1] + 2 = M(n-3) + 3.
\end{aligned}
$$

After inspecting the first three lines, we see an emerging pattern, which makes it possible to predict not only the next line (what would it be?) but also a general formula for the pattern: $M(n) = M(n - i) + i$. Strictly speaking, the correctness of this formula should be proved by mathematical induction, but it is easier to get to the solution as follows and then verify its correctness.

What remains to be done is to take advantage of the initial condition given. Since it is specified for $n = 0$, we have to substitute $i = n$ in the pattern's formula to get the ultimate result of our backward substitutions:

$$M(n) = M(n-1) + 1 = \cdots = M(n-i) + i = \cdots = M(n-n) + n = n.$$

You should not be disappointed after exerting so much effort to get this "obvious" answer. The benefits of the method illustrated in this simple example will become clear very soon, when we have to solve more difficult recurrences. Also, note that the simple iterative algorithm that accumulates the product of n consecutive integers requires the same number of multiplications, and it does so without the overhead of time and space used for maintaining the recursion's stack.

The issue of time efficiency is actually not that important for the problem of computing $n!$, however. As we saw in Section 2.1, the function's values get so large so fast that we can realistically compute exact values of $n!$ only for very small n's. Again, we use this example just as a simple and convenient vehicle to introduce the standard approach to analyzing recursive algorithms.

Generalizing our experience with investigating the recursive algorithm for computing $n!$, we can now outline a general plan for investigating recursive algorithms.

General Plan for Analyzing the Time Efficiency of Recursive Algorithms

Decide on a parameter (or parameters) indicating an input's size.

Identify the algorithm's basic operation.

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.

Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.

Solve the recurrence or, at least, ascertain the order of growth of its solution.

EXAMPLE 2 As our next example, we consider another educational workhorse of recursive algorithms: the ***Tower of Hanoi*** puzzle. In this puzzle, we (or mythical monks, if you do not like to move disks) have n disks of different sizes that can slide onto any of three pegs. Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on top. The goal is to move all the disks to the third peg, using the second one as an auxiliary, if necessary. We can move only one disk at a time, and it is forbidden to place a larger disk on top of a smaller one.

The problem has an elegant recursive solution, which is illustrated in Fig-ure 2.4. To move $n > 1$ disks from peg 1 to peg 3 (with peg 2 as auxiliary), we first move recursively $n - 1$ disks from peg 1 to peg 2 (with peg 3 as auxiliary), then move the largest disk directly from peg 1 to peg 3, and, finally, move recursively $n - 1$ disks from peg 2 to peg 3 (using peg 1 as auxiliary). Of course, if $n = 1$**,** we simply move the single disk directly from the source peg to the destination peg.

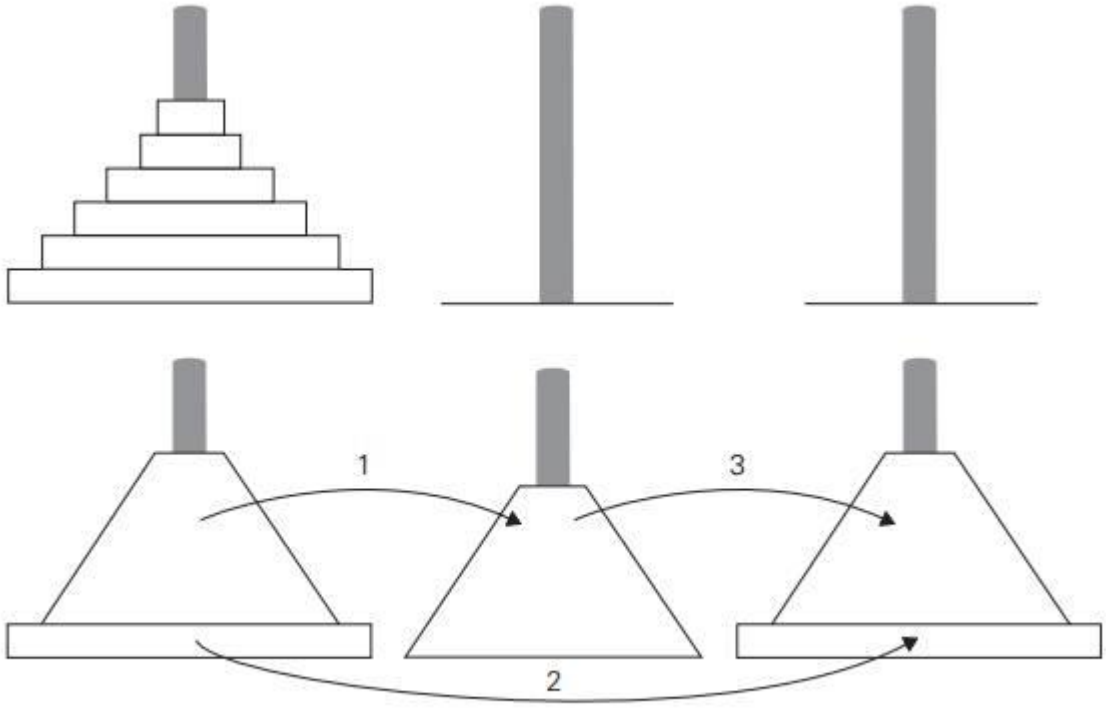


FIGURE 2.4 Recursive solution to the Tower of Hanoi puzzle.

Let us apply the general plan outlined above to the Tower of Hanoi problem. The number of disks ***n*** is the obvious choice for the input's size indicator, and so is moving one disk as the algorithm's basic operation. Clearly, the number of moves ***M(n)*** depends on ***n*** only, and we get the following recurrence equation for it:

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

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

$$M(n) = 2M(n-1) + 1 \quad \text{for } n > 1, \qquad \textbf{(2.3)}$$
$$M(1) = 1.$$

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

$$M(n) = 2M(n-1) + 1 \qquad \text{sub. } M(n-1) = 2M(n-2) + 1$$
$$= 2[2M(n-2) + 1] + 1 = 2^2 M(n-2) + 2 + 1 \quad \text{sub. } M(n-2) = 2M(n-3) + 1$$
$$= 2^2[2M(n-3) + 1] + 2 + 1 = 2^3 M(n-3) + 2^2 + 2 + 1.$$

The pattern of the first three sums on the left suggests that the next one will be $2^4 M(n-4) + 2^3 + 2^2 + 2 + 1$, and generally, after i substitutions, we get

$$M(n) = 2^i M(n-i) + 2^{i-1} + 2^{i-2} + \cdots + 2 + 1 = 2^i M(n-i) + 2^i - 1.$$

Since the initial condition is specified for $n = 1$, which is achieved for $i = n - 1$, we get the following formula for the solution to recurrence (2.3):

$$M(n) = 2^{n-1} M(n - (n-1)) + 2^{n-1} - 1$$
$$= 2^{n-1} M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1.$$

Thus, we have an exponential algorithm, which will run for an unimaginably long time even for moderate values of ***n*** (see Problem 5 in this section's exercises). This is not due to the fact that this particular algorithm is poor; in fact, it is not difficult to prove that this is the most efficient algorithm possible for this problem. It is the problem's intrinsic difficulty that makes it so computationally hard. Still, this example makes an important general point:

One should be careful with recursive algorithms because their succinctness may mask their inefficiency.

Time Complexity and Space Complexity

Time Complexity: The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. Note that the time to run is a function of the length of the input and not the actual execution time of the machine on which the algorithm is running on.
Definition–

The valid algorithm takes a finite amount of time for execution. The time required by the algorithm to solve given problem is called ***time complexity*** of the algorithm. Time complexity is very useful measure in algorithm analysis. It is the time needed for the completion of an algorithm. To estimate the time complexity, we need to consider the cost of each fundamental instruction and the number of times the instruction is executed.
Example 1: Addition of two scalar variables.

```
Algorithm ADD SCALAR(A, B)
//Description: Perform arithmetic addition of two numbers
//Input: Two scalar variables A and B
//Output: variable C, which holds the addition of A and B
C <- A + B
return C
```

The addition of two scalar numbers requires one addition operation. the time complexity of this algorithm is constant, so T(n) = O(1) .

Space Complexity:

Definition –
Problem-solving using computer requires memory to hold temporary data or final result while the program is in execution. The amount of memory required by the algorithm to solve given problem is called **space complexity** of the algorithm.
The space complexity of an algorithm quantifies the amount of space taken by an algorithm to run as a function of the length of the input. Consider an example: Suppose a problem to find the frequency of array elements.
It is the amount of memory needed for the completion of an algorithm.
To estimate the memory requirement we need to focus on two parts:
(1) A fixed part: It is independent of the input size. It includes memory for instructions (code), constants, variables, etc.
(2) A variable part: It is dependent on the input size. It includes memory for recursion stack, referenced variables, etc.