DATA STRUCTURES AND ALGORITHMS MADE EASY

INTRODUCTION

Chapter-1

The objective of this chapter is to explain the importance of analysis of algorithms, their notations, relationships and solving as many problems as possible. We first concentrate on understanding the basic elements of algorithms, importance of analysis and then slowly move towards analyzing the algorithms with different notations and finally the problems. After completion of this chapter you should be able to find the complexity of any given algorithm (especially recursive functions).

1.1 Variables

Before going to the definition of variables, let us relate them to old mathematical equations. All of us have solved many mathematical equations since childhood. As an example, consider the below equation:

$$x^2 + 2y - 2 = 1$$

We don't have to worry about the use of above equation. The important thing that we need to understand is, the equation has some names (x and y) which hold values (data). That means, the *names* (x and y) are the place holders for representing data. Similarly, in computer science we need something for holding data and *variables* are the facility for doing that.

1.2 Data types

In the above equation, the variables x and y can take any values like integral numbers (10, 20 etc...), real numbers (0.23, 5.5 etc...) or just 0 and 1. To solve the equation, we need to relate them to kind of values they can take and *data type* is the name being used in computer science for this purpose.

A *data type* in a programming language is a set of data with values having predefined characteristics. Examples of data types are: integer, floating point unit number, character, string etc...

Computer memory is all filled with zeros and ones. If we have a problem and wanted to code it, it's very difficult to provide the solution in terms of zeros and ones. To help users, programming languages and compilers are providing the facility of data types.

For example, *integer* takes 2 bytes (actual value depends on compiler), *float* takes 4 bytes etc... This says that, in memory we are combining 2 bytes (16 bits) and calling it as *integer*. Similarly, combining 4 bytes (32 bits) and calling it as *float*. A data type reduces the coding effort. Basically, at the top level, there are two types of data types:
- System defined data types (also called *Primitive* data types)
- User defined data types

System defined data types (Primitive data types)

Data types which are defined by system are called *primitive* data types. The primitive data types which are provided by many programming languages are: int, float, char, double, bool, etc...

The number of bits allocated for each primitive data type depends on the programming languages, compiler and operating system. For the same primitive data type, different languages may use different sizes. Depending on the size of the data types the total available values (domain) will also changes. For example, "*int*" may take 2 bytes or 4 bytes. If it takes 2 bytes (16 bits) then the total possible values are -32,768 to +32,767 (-2^{15} *to* $2^{15}-1$). If it takes, 4 bytes (32 bits), then the possible values are between −2,147,483,648 to +2,147,483,648 (-2^{31} *to* $2^{31}-1$). Same is the case with remaining data types too.

User defined data types

If the system defined data types are not enough then most programming languages allows the users to define their own data types called as user defined data types. Good example of user defined data types are: structures in $C/C++$ and classes in $Java$.

For example, in the below case, we are combining many system defined data types and called it as user defined data type with name "*newType*". This gives more flexibility and comfort in dealing with computer memory.

```
struct newType {
        int data1;
        float data 2;
        ...
        char data;
};
```

1.3 Data Structure

Based on the above discussion, once we have data in variables, we need some mechanism for manipulating that data to solve problems. *Data structure* is a particular way of storing and organizing data in a computer so that it can be used efficiently. That means, a *data structure* is a specialized format for organizing and storing data. General data structure types include arrays, files, linked lists, stacks, queues, trees, graphs and so on.

Depending on the organization of the elements, data structures are classified into two types:

1) *Linear data structures*: Elements are accessed in a sequential order but it is not compulsory to store all elements sequentially (say, Linked Lists). *Examples*: Linked Lists, Stacks and Queues.
2) *Non – linear data structures*: Elements of this data structure are stored/accessed in a non-linear order. *Examples*: Trees and graphs.

1.4 Abstract Data Types (ADTs)

Before defining abstract data types, let us consider the different view of system defined data types. We all know that, by default, all primitive data types (int, float, et..) supports basic operations like addition, subtraction etc... The system is providing the implementations for the primitive data types. For user defined data types also we need to define operations. The implementation for these operations can be done when we want to actually use them. That means, in general user defined data types are defined along with their operations.

To simplify the process of solving the problems, we generally combine the data structures along with their operations and are called *Abstract Data Types* (ADTs). An ADT consists of *two* parts:

1. Declaration of data
2. Declaration of operations

Commonly used ADTs *include*: Linked Lists, Stacks, Queues, Priority Queues, Binary Trees, Dictionaries, Disjoint Sets (Union and Find), Hash Tables, Graphs, and many other. For example, stack uses LIFO (Last-In-First-Out) mechanism while storing the data in data structures. The last element inserted into the stack is the first element that gets deleted. Common operations of it are: creating the stack, pushing an element onto the stack, popping an element from stack, finding the current top of the stack, finding number of elements in the stack etc...

While defining the ADTs do not care about implementation details. They come in to picture only when we want to use them. Different kinds of ADTs are suited to different kinds of applications, and some are highly specialized to specific tasks. By the end of this book, we will go through many of them and you will be in a position to relate the data structures to the kind of problems they solve.

1.5 What is an Algorithm?

Let us consider the problem of preparing an omelet. For preparing omelet, general steps we follow are:

1) Get the frying pan.
2) Get the oil.
    a. Do we have oil?
        i. If yes, put it in the pan.
        ii. If no, do we want to buy oil?
            1. If yes, then go out and buy.
            2. If no, we can terminate.
3) Turn on the stove, etc...

What we are doing is, for a given problem (preparing an omelet), giving step by step procedure for solving it. Formal definition of an algorithm can be given as:

An algorithm is the step-by-step instructions to solve a given problem.

Note: we do not have to prove each step of the algorithm.

1.6 Why Analysis of Algorithms?

To go from city "*A*" to city "*B*", there can be many ways of accomplishing this: by flight, by bus, by train and also by cycle. Depending on the availability and convenience we choose the one which suits us. Similarly, in computer science there can be multiple algorithms exist for solving the same problem (for example, sorting problem has many algorithms like insertion sort, selection sort, quick sort and many more). Algorithm analysis helps us determining which of them is efficient in terms of time and space consumed.

1.7 Goal of Analysis of Algorithms

The goal of *analysis of algorithms* is to compare algorithms (or solutions) mainly in terms of running time but also in terms of other factors (e.g., memory, developers effort etc.)

1.8 What is Running Time Analysis?

It is the process of determining how processing time increases as the size of the problem (input size) increases. Input size is number of elements in the input and depending on the problem type the input may be of different types. In general, we encounter the following types of inputs.

- Size of an array

- Polynomial degree
- Number of elements in a matrix
- Number of bits in binary representation of the input
- Vertices and edges in a graph

1.9 How to Compare Algorithms?

To compare algorithms, let us define few *objective measures*:
Execution times? *Not a good measure* as execution times are specific to a particular computer.

Number of statements executed? *Not a good measure*, since the number of statements varies with the programming language as well as the style of the individual programmer.

Ideal Solution? Let us assume that we expressed running time of given algorithm as a function of the input size n (i.e., $f(n)$) and compare these different functions corresponding to running times. This kind of comparison is independent of machine time, programming style, etc...

1.10 What is Rate of Growth?

The rate at which the running time increases as a function of input is called *rate of growth*. Let us assume that you went to a shop for buying a car and a cycle. If your friend sees you there and asks what you are buying then in general we say *buying a car*. This is because, cost of car is too big compared to cost of cycle (approximating the cost of cycle to cost of car).

$$Total\ Cost\ =\ cost\_of\_car\ +\ cost\_of\_cycle$$
$$Total\ Cost\ \approx\ cost\_of\_car\ (approximation)$$

For the above example, we can represent the cost of car and cost of cycle in terms of function and for a given function ignore the low order terms that are relatively insignificant (for large value of input size, n). As an example in the below case, n^4, $2n^2$, $100n$ and 500 are the individual costs of some function and approximate it to n^4. Since, n^4 is the highest rate of growth.

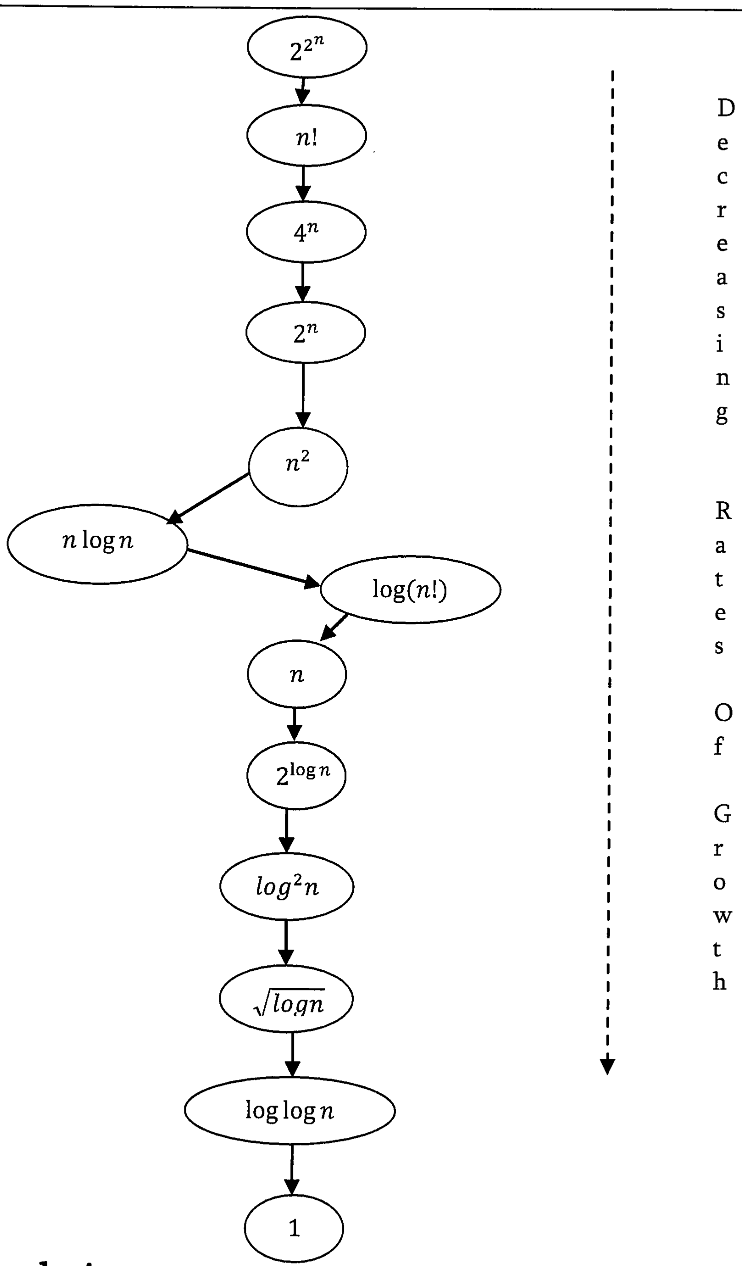
$$n^4\ +\ 2n^2\ +\ 100n\ +\ 500\ \approx\ n^4$$

1.11 Commonly used Rate of Growths

Below is the list of rate of growths which come across in remaining chapters.

Time complexity	Name	Example
1	Constant	Adding an element to the front of a linked list
$logn$	Logarithmic	Finding an element in a sorted array
n	Linear	Finding an element in an unsorted array
$nlogn$	Linear Logarithmic	Sorting n items by 'divide-and-conquer'-Mergesort
n^2	Quadratic	Shortest path between two nodes in a graph
n^3	Cubic	Matrix Multiplication
2^n	Exponential	The Towers of Hanoi problem

Below diagram shows the relationship between different rates of growth.



1.12 Types of Analysis

To analyze the given algorithm we need to know on what inputs the algorithm is taking less time (performing well) and on what inputs the algorithm is taking huge time. We have already seen that an algorithm can be represented in the form of an expression. That means we represent the algorithm with multiple expressions: one for case where it is taking the less time and other for case where it is taking the more time. In general the first case is called the *best case* and second case is called the *worst case* of the algorithm. To analyze an algorithm we need some kind of syntax and that forms the base for asymptotic analysis/notation. There are three types of analysis:

- Worst case
  - Defines the input for which the algorithm takes huge time.
  - Input is the one for which the algorithm runs the slower.
- Best case
  - Defines the input for which the algorithm takes lowest time.

    - Input is the one for which the algorithm runs the fastest.
- Average case
    - Provides a prediction about the running time of the algorithm
    - Assumes that the input is random

$$Lower\ Bound <= Average\ Time <= Upper\ Bound$$

For a given algorithm, we can represent best, worst and average cases in the form of expressions. As an example, let $f(n)$ be the function which represents the given algorithm.

$$f(n) = n^2 + 500, \text{ for worst case}$$
$$f(n) = n + 100n + 500, \text{ for best case}$$

Similarly, for average case too. The expression defines the inputs with which the algorithm takes the average running time (or memory).

1.13 Asymptotic Notation

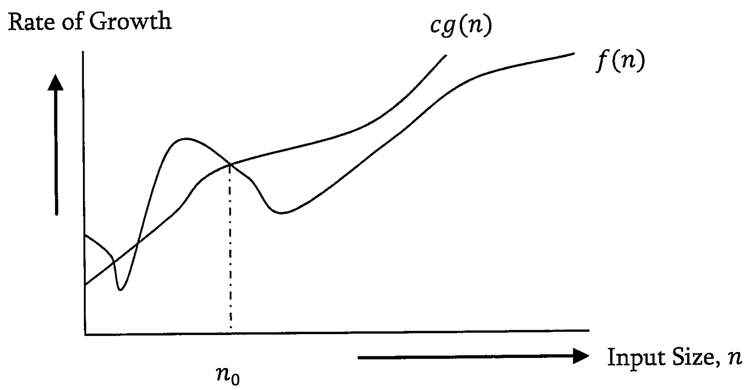
Having the expressions for best, average case and worst cases, for all the three cases we need to identify the upper and lower bounds. In order to represent these upper and lower bounds we need some kind syntax and that is the subject of following discussion. Let us assume that the given algorithm is represented in the form of function $f(n)$.

1.14 Big-O Notation

This notation gives the *tight* upper bound of the given function. Generally, it is represented as $f(n) = O(g(n))$. That means, at larger values of n, the upper bound of $f(n)$ is $g(n)$. For example, if $f(n) = n^4 + 100n^2 + 10n + 50$ is the given algorithm, then n^4 is $g(n)$. That means, $g(n)$ gives the maximum rate of growth for $f(n)$ at larger values of n.

Let us see the O–notation with little more detail. O–notation defined as $O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0\}$. $g(n)$ is an asymptotic tight upper bound for $f(n)$. Our objective is to give smallest rate of growth $g(n)$ which is greater than or equal to given algorithms rate of growth $f(n)$.

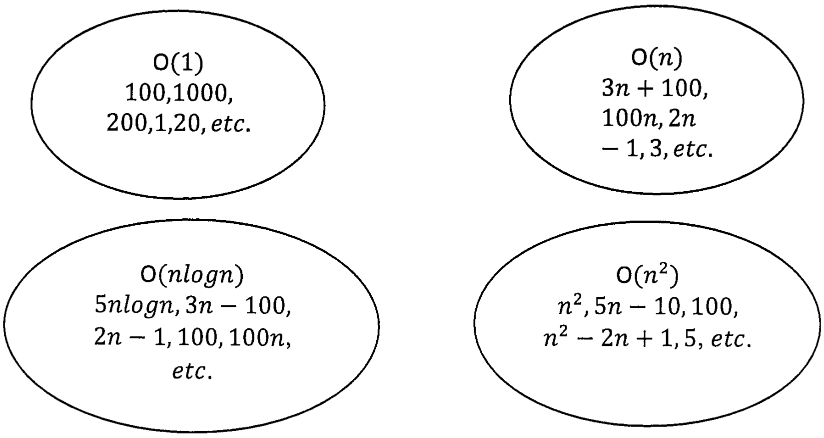
In general, we discard lower values of n. That means the rate of growth at lower values of n is not important. In the below figure, n_0 is the point from which we need to consider the rate of growths for a given algorithm. Below n_0 the rate of growths could be different.



Big-O Visualization

$O(g(n))$ is the set of functions with smaller or same order of growth as $g(n)$. For example, $O(n^2)$ includes $O(1)$, $O(n)$, $O(nlogn)$ etc..

Note: Analyze the algorithms at larger values of n only. What this means is, below n_0 we do not care for rate of growth.



Big-O Examples

Example-1 Find upper bound for $f(n) = 3n + 8$
Solution: $3n + 8 \leq 4n$, for all $n \geq 1$
$\therefore 3n + 8 = O(n)$ with c = 4 and $n_0 = 8$

Example-2 Find upper bound for $f(n) = n^2 + 1$
Solution: $n^2 + 1 \leq 2n^2$, for all $n \geq 1$
$\therefore n^2 + 1 = O(n^2)$ with $c = 2$ and $n_0 = 1$

Example-3 Find upper bound for $f(n) = n^4 + 100n^2 + 50$
Solution: $n^4 + 100n^2 + 50 \leq 2n^4$, for all $n \geq 11$
$\therefore n^4 + 100n^2 + 50 = O(n^4)$ with $c = 2$ and $n_0 = 11$

Example-4 Find upper bound for $f(n) = 2n^3 - 2n^2$
Solution: $2n^3 - 2n^2 \leq 2n^3$, for all $n \geq 1$
$\therefore 2n^3 - 2n^2 = O(2n^3)$ with $c = 2$ and $n_0 = 1$

Example-5 Find upper bound for $f(n) = n$
Solution: $n \leq n^2$, for all $n \geq 1$
$\therefore n = O(n^2)$ with $c = 1$ and $n_0 = 1$

Example-6 Find upper bound for $f(n) = 410$
Solution: $410 \leq 410$, for all $n \geq 1$
$\therefore 100 = O(1)$ with $c = 1$ and $n_0 = 1$

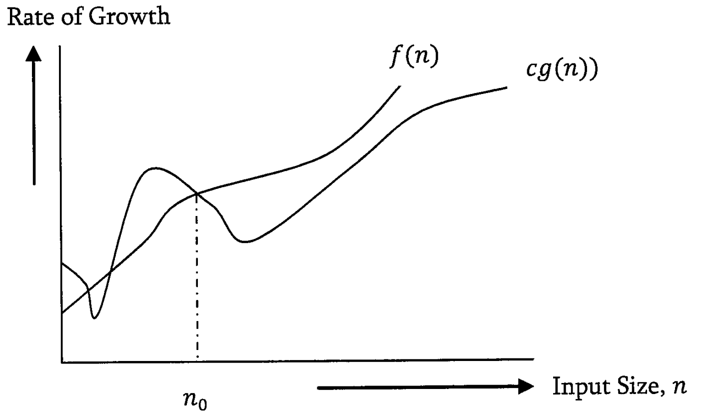
No Uniqueness?

There are no unique set of values for n_0 and c in proving the asymptotic bounds. Let us consider, $100n + 5 = O(n^2)$. For this function there are multiple n_0 and c values possible.

Solution1: $100n + 5 \leq 100n + n = 101n \leq 101n^2$ for all $n \geq 5$, $n_0 = 5$ and $c = 101$ is a solution.
Solution2: $100n + 5 \leq 100n + 5n = 105n \leq 105n^2$ for all $n \geq 1$, $n_0 = 1$ and $c = 105$ is also a solution.

1.15 Omega-Ω Notation

Similar to O discussion, this notation gives the tighter lower bound of the given algorithm and we represent it as $f(n) = \Omega(g(n))$. That means, at larger values of n, the tighter lower bound of $f(n)$ is $g(n)$. For example, if $f(n) = 100n^2 + 10n + 50$, $g(n)$ is $\Omega(n^2)$.



The Ω notation can be defined as $\Omega(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0\}$. $g(n)$ is an asymptotic tight lower bound for $f(n)$. Our objective is to give largest rate of growth $g(n)$ which is less than or equal to given algorithms rate of growth $f(n)$.

Ω Examples

Example-1 Find lower bound for $f(n) = 5n^2$

Solution: $\exists\, c,\, n_0$ Such that: $0 \leq cn \leq 5n^2 \Rightarrow cn \leq 5\,n^2 \Rightarrow c = 1$ and $n_0 = 1$

$\therefore 5n^2 = \Omega(n)$ with $c = 1$ and $n_0 = 1$

Example-2 Prove $f(n) = 100n + 5 \neq \Omega(n^2)$

Solution: $\exists$ c, n_0 Such that: $0 \leq cn^2 \leq 100n + 5$

$100n + 5 \leq 100n + 5n\ (\forall n \geq 1) = 105n$

$cn^2 \leq 105n \Rightarrow n(cn - 105) \leq 0$

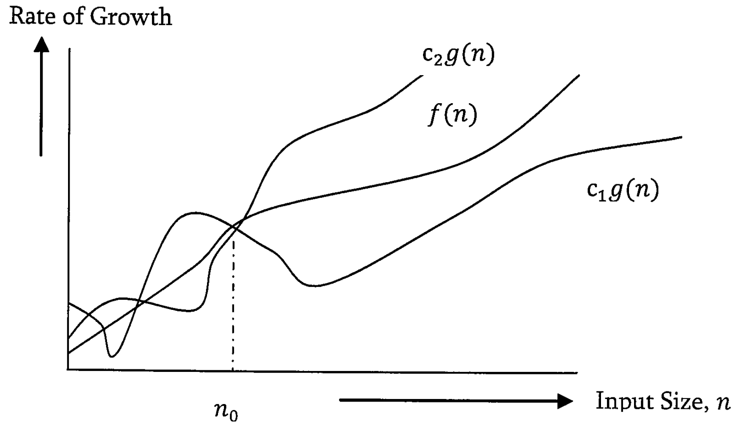
Since n is positive $\Rightarrow cn - 105 \leq 0 \Rightarrow n \leq 105/c$

$\Rightarrow$ Contradiction: n cannot be smaller than a constant

Example-3 $n = \Omega(2n)$, $n^3 = \Omega(n^2)$, $n = \Omega(logn)$

1.16 Theta-Θ Notation

This notation decides whether the upper and lower bounds of a given function (algorithm) are same or not. The average running time of algorithm is always between lower bound and upper bound. If the upper bound (O) and lower bound (Ω) gives the same result then Θ notation will also have the same rate of growth. As an example, let us assume that $f(n) = 10n + n$ is the expression. Then, its tight upper bound $g(n)$ is $O(n)$. The rate of growth in best case is $g(n) = O(n)$. In this case, rate of growths in best case and worst are same. As a result, the average case will also be same. For a given function (algorithm), if the rate of growths (bounds) for O and Ω are not same then the rate of growth Θ case may not be same.



Now consider the definition of Θ notation. It is defined as $\Theta(g(n)) = \{f(n)$: there exist positive constants c_1, c_2 and n_0 such that $0 \leq c_1g(n) \leq f(n) \leq c_2g(n)$ for all $n \geq n_0\}$. $g(n)$ is an asymptotic tight bound for $f(n)$. $\Theta(g(n))$ is the set of functions with the same order of growth as $g(n)$.

Θ Examples

Example-1 Find Θ bound for $f(n) = \frac{n^2}{2} - \frac{n}{2}$
Solution: $\frac{n^2}{5} \leq \frac{n^2}{2} - \frac{n}{2} \leq n^2$, for all, $n \geq 1$
$\therefore \frac{n^2}{2} - \frac{n}{2} = \Theta(n^2)$ with $c_1 = 1/5, c_2 = 1$ and $n_0 = 1$

Example-2 Prove $n \neq \Theta(n^2)$
Solution: $c_1 n^2 \leq n \leq c_2n^2 \Rightarrow$ only holds for: $n \leq 1/c_1$
$\therefore n \neq \Theta(n^2)$

Example-3 Prove $6n^3 \neq \Theta(n^2)$
Solution: $c_1 n^2 \leq 6n^3 \leq c_2 n^2 \Rightarrow$ only holds for: $n \leq c_2 /6$
$\therefore 6n^3 \neq \Theta(n^2)$

Example-4 Prove $n \neq \Theta(logn)$
Solution: $c_1 logn \leq n \leq c_2 \, log\, n \Rightarrow c_2 \geq \frac{n}{\log n}, \forall\, n \geq n_0$ – Impossible

Important Notes

For analysis (best case, worst case and average) we try to give upper bound (O) and lower bound (Ω) and average running time (Θ). From the above examples, it should also be clear that, for a given function (algorithm) getting upper bound (O) and lower bound (Ω) and average running time (Θ) may not be possible always. For example, if we are discussing the best case of an algorithm, then we try to give upper bound (O) and lower bound (Ω) and average running time (Θ). In the remaining chapters we generally concentrate on upper bound (O) because knowing lower bound (Ω) of an algorithm is of no practical importance and we use Θ notation if upper bound (O) and lower bound (Ω) are same.

1.17 Why is it called Asymptotic Analysis?

From the above discussion (for all the three notations: worst case, best case and average case), we can easily understand that, in every case for a given function $f(n)$ we are trying to find other function $g(n)$ which approximates $f(n)$ at higher values of n. That means, $g(n)$ is also a curve which approximates $f(n)$ at higher values of n. In mathematics we call such curve as *asymptotic curve*. In other terms, $g(n)$ is the asymptotic curve for $f(n)$. For this reason, we call algorithm analysis as *asymptotic analysis*.

1.18 Guidelines for Asymptotic Analysis

There are some general rules to help us in determining the running time of an algorithm.

1) **Loops**: The running time of a loop is, at most, the running time of the statements inside the loop (including tests) multiplied by the number of iterations.

```
// executes n times
for (i=1; i<=n; i++)
    m = m + 2; // constant time, c
```

Total time = a constant $c \times n = c\,n = O(n)$.

2) **Nested loops:** Analyze from inside out. Total running time is the product of the sizes of all the loops.

```
//outer loop executed n times
for (i=1; i<=n; i++) {
    // inner loop executed n times
    for (j=1; j<=n; j++)
            k = k+1; //constant time
}
```

Total time = $c \times n \times n = cn^2 = O(n^2)$.

3) **Consecutive statements:** Add the time complexities of each statement.

```
x = x +1; //constant time
// executed n times
for (i=1; i<=n; i++)
    m = m + 2; //constant time
//outer loop executed n times
for (i=1; i<=n; i++) {
    //inner loop executed n times
    for (j=1; j<=n; j++)
            k = k+1; //constant time
}
```

Total time = $c_0 + c_1 n + c_2 n^2 = O(n^2)$.

4) **If-then-else statements:** Worst-case running time: the test, plus *either* the *then* part *or* the *else* part (whichever is the larger).

```
//test: constant
if(length( ) == 0 ) {
    return false; //then part: constant
}
else {// else part: (constant + constant) * n
    for (int n = 0; n < length( ); n++) {
            // another if : constant + constant (no else part)
            if(!list[n].equals(otherList.list[n]))
                    //constant
                    return false;
    }
}
```

Total time = $c_0 + c_1 + (c_2 + c_3) * n = O(n)$.

5) **Logarithmic complexity:** An algorithm is $O(logn)$ if it takes a constant time to cut the problem size by a fraction (usually by ½). As an example let us consider the following program:

```
for (i=1; i<=n;)
        i = i*2;
```

If we observe carefully, the value of i is doubling every time. Initially $i = 1$, in next step $i = 2$, and in subsequent steps $i = 4, 8$ and so on. Let us assume that the loop is executing some k times. At k^{th} step $2^i = n$ and we come out of loop. Taking logarithm on both sides, gives

$log(2^i) = logn$
$ilog2 = logn$
$i = logn$ //if we assume base-2

Total time = $O(logn)$.

Note: Similarly, for the below case also, worst case rate of growth is $O(logn)$. The same discussion holds good for decreasing sequence as well.

```
for (i=n; i<=1;)
    i = i/2;
```

Another example: binary search (finding a word in a dictionary of n pages)

- Look at the center point in the dictionary
- Is word towards left or right of center?
- Repeat process with left or right part of dictionary until the word is found

1.19 Properties of Notations

- Transitivity: $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$. Valid for O and Ω as well.
- Reflexivity: $f(n) = \Theta(f(n))$. Valid for O and Ω also.
- Symmetry: $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.
- Transpose symmetry: $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.

1.20 Commonly used Logarithms and Summations

Logarithms

$log\, x^y = y\, log\, x$ $\qquad logn = log_{10}^{n}$
$log\, xy = logx + logy$ $\qquad log^k n = (logn)^k$
$log\, logn = log(logn)$ $\qquad log\frac{x}{y} = logx - logy$
$a^{log_b^x} = x^{log_b^a}$ $\qquad log_b^x = \frac{log_a^x}{log_a^b}$

Arithmetic series

$$\sum_{K=1}^{n} k = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

Geometric series

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 \ldots + x^n = \frac{x^{n+1} - 1}{x - 1} (x \neq 1)$$

Harmonic series

$$\sum_{k=1}^{n} \frac{1}{k} = 1 + \frac{1}{2} + \ldots + \frac{1}{n} \approx log\, n$$

Other important formulae

$$\sum_{k=1}^{n} log\, k \approx nlogn$$

$$\sum_{k=1}^{n} k^p = 1^p + 2^p + \cdots + n^p \approx \frac{1}{p+1} n^{p+1}$$

1.21 Master Theorem for Divide and Conquer

All divide and conquer algorithms (In detail, we will discuss them in *Divide and Conquer* chapter) divides the problem into subproblems, each of which is part of the original problem, and then perform some additional work to compute the final answer. As an example, merge sort algorithm [for details, refer *Sorting* chapter] operates on two subproblems, each of which is half the size of the original and then performs $O(n)$ additional work for merging. This gives the running time equation:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

The following theorem can be used to determine the running time of divide and conquer algorithms. For a given program (algorithm), first we try to find the recurrence relation for the problem. If the recurrence is of the below form then we can directly give the answer without fully solving it.

If the recurrence is of the form $T(n) = aT(\frac{n}{b}) + \Theta(n^k log^p n)$, where $a \geq 1$, $b > 1, k \geq 0$ and p is a real number, then:

1) If $a > b^k$, then $T(n) = \Theta(n^{log_b^a})$
2) If $a = b^k$
   a. If $p > -1$, then $T(n) = \Theta(n^{log_b^a} log^{p+1} n)$
   b. If $p = -1$, then $T(n) = \Theta(n^{log_b^a} loglogn)$
   c. If $p < -1$, then $T(n) = \Theta(n^{log_b^a})$
3) If $a < b^k$
   a. If $p \geq 0$, then $T(n) = \Theta(n^k log^p n)$
   b. If $p < 0$, then $T(n) = O(n^k)$

1.22 Problems on Divide and Conquer Master Theorem

For each of the following recurrences, give an expression for the runtime $T(n)$ if the recurrence can be solved with the Master Theorem. Otherwise, indicate that the Master Theorem does not apply.

Problem-1 $T(n) = 3T\ (n/2) + n^2$

Solution: $T(n) = 3T\ (n/2) + n^2 \Rightarrow T\ (n) = \Theta(n^2)$ (Master Theorem Case 3.a)

Problem-2 $T(n) = 4T\ (n/2) + n^2$

Solution: $T(n) = 4T\ (n/2) + n^2 \Rightarrow T\ (n) = \Theta(n^2 logn)$ (Master Theorem Case 2.a)

Problem-3 $T(n) = T(n/2) + n^2$

Solution: $T(n) = T(n/2) + n^2 \Rightarrow \Theta(n^2)$ (Master Theorem Case 3.a)

Problem-4 $T(n) = 2^n T(n/2) + n^n$

Solution: $T(n) = 2^n T(n/2) + n^n$ => Does not apply (a is not constant)

Problem-5 $T(n) = 16T(n/4) + n$

Solution: $T(n) = 16T(n/4) + n \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 1)

Problem-6 $T(n) = 2T(n/2) + nlogn$

Solution: $T(n) = 2T(n/2) + nlogn \Rightarrow T(n) = \Theta(nlog^2 n)$ (Master Theorem Case 2.a)

Problem-7 $T(n) = 2T(n/2) + n/logn$

Solution: $T(n) = 2T(n/2) + n/logn \Rightarrow T(n) = \Theta(nloglogn)$ (Master Theorem Case 2.b)

Problem-8 $T(n) = 2T(n/4) + n^{0.51}$

Solution: $T(n) = 2T(n/4) + n^{0.51} \Rightarrow T(n) = O(n^{0.51})$ (Master Theorem Case 3.b)

Problem-9 $T(n) = 0.5T(n/2) + 1/n$

Solution: $T(n) = 0.5T(n/2) + 1/n \Rightarrow$ Does not apply ($a < 1$)

Problem-10 $T(n) = 6T(n/3) + n^2 logn$

Solution: $T(n) = 6T(n/3) + n^2 logn \Rightarrow T(n) = \Theta(n^2 logn)$ (Master Theorem Case 3.a)

Problem-11 $T(n) = 64T(n/8) - n^2 logn$

Solution: $T(n) = 64T(n/8) - n^2 logn \Rightarrow$ Does not apply (function is not positive)

Problem-12 $T(n) = 7T(n/3) + n^2$

Solution: $T(n) = 7T(n/3) + n^2 \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 3.as)

Problem-13 $T(n) = 4T(n/2) + logn$

Solution: $T(n) = 4T(n/2) + logn \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 1)

Problem-14 $T(n) = 16T(n/4) + n!$

Solution: $T(n) = 16T(n/4) + n! \Rightarrow T(n) = \Theta(n!)$ (Master Theorem Case 3.a)

Problem-15 $T(n) = \sqrt{2}T(n/2) + logn$

Solution: $T(n) = \sqrt{2}T(n/2) + logn \Rightarrow T(n) = \Theta(\sqrt{n})$ (Master Theorem Case 1)

Problem-16 $T(n) = 3T(n/2) + n$

Solution: $T(n) = 3T(n/2) + n \Rightarrow T(n) = \Theta(n^{log3})$ (Master Theorem Case 1)

Problem-17 $T(n) = 3T(n/3) + \sqrt{n}$

Solution: $T(n) = 3T(n/3) + \sqrt{n} \Rightarrow T(n) = \Theta(n)$ (Master Theorem Case 1)

Problem-18 $T(n) = 4T(n/2) + cn$

Solution: $T(n) = 4T(n/2) + cn \Rightarrow T(n) = \Theta(n^2)$ (Master Theorem Case 1)

Problem-19 $T(n) = 3T(n/4) + nlogn$

Solution: $T(n) = 3T(n/4) + nlogn \Rightarrow T(n) = \Theta(nlogn)$ (Master Theorem Case 3.a)

Problem-20 $T(n) = 3T(n/3) + n/2$

Solution: $T(n) = 3T(n/3) + n/2 \Rightarrow T(n) = \Theta(nlogn)$ (Master Theorem Case 2.a)

1.23 Master Theorem for Subtract and Conquer Recurrences

Let $T(n)$ be a function defined on positive n, and having the property

$$T(n) = \begin{cases} c, & \text{if } n \leq 1 \\ aT(n-b) + f(n), & \text{if } n > 1 \end{cases}$$

for some constants $c, a > 0, b > 0, k \geq 0$, and function $f(n)$. If $f(n)$ is in $O(n^k)$, then

$$T(n) = \begin{cases} O(n^k), & \text{if a} < 1 \\ O(n^{k+1}), & \text{if a} = 1 \\ O\left(n^k a^{\frac{n}{b}}\right), & \text{if a} > 1 \end{cases}$$

1.24 Variant of subtraction and conquer master theorem

The solution to the equation $T(n) = T(\alpha\, n) + T((1-\alpha)n) + \beta n$, where $0 < \alpha < 1$ and $\beta > 0$ are constants, is $O(nlogn)$.

1.25 Amortized Analysis

Amortized analysis refers to determining the time-averaged running time for a sequence of operations. It is different from average case analysis, because amortized analysis does not make any assumption about the distribution of the data values, whereas average case analysis assumes the data are not "bad" (e.g., some sorting algorithms do well on "average" over all input orderings but very badly on certain input orderings). That is, amortized analysis is a worst case analysis, but for a sequence of operations, rather than for individual operations.

The motivation for amortized analysis is to better understand the running time of certain techniques, where standard worst case analysis provides an overly pessimistic bound. Amortized analysis generally applies to a method that consists of a sequence of operations, where the vast majority of the operations are cheap, but some of the operations are expensive. If we can show that the expensive operations are particularly rare we can "charge them" to the cheap operations, and only bound the cheap operations.

The general approach is to assign an artificial cost to each operation in the sequence, such that the total of the artificial costs for the sequence of operations bounds total of the real costs for the sequence. This artificial cost is called the amortized cost of an operation. In order to analyze the running time, the amortized cost thus is a correct way of understanding the overall running time — but note that particular operations can still take longer so it is not a way of bounding the running time of any individual operation in the sequence.

When one event in a sequence affects the cost of later events:

- One particular task may be expensive.
- But it may leave data structure in a state that next few operations becomes easier.

Example: Let us consider an array of elements from which we want to find k^{th} smallest element. We can solve this problem using sorting. After sorting the given array, we just need to return the k^{th} element from it. Cost of performing sort (assuming comparison based sorting algorithm) is $O(nlogn)$. If we perform n such selections then the average cost of each selection is $O(nlogn/n) = O(logn)$. This clearly indicates that sorting once is reducing the complexity of subsequent operations.

1.26 Problems on Algorithms Analysis

Note: From the following problems, try to understand the cases which give different complexities ($O(n)$, $O(logn)$, $O(loglogn)$ etc...).

Problem-21 Find the complexity of the below recurrence:

$$T(n) = \begin{cases} 3T(n-1), if\ n > 0, \\ 1, \quad otherwise \end{cases}$$

Solution: Let us try solving this function with substitution.

$$T(n) = 3T(n-1)$$
$$T(n) = 3(3T(n-2)) = 3^2T(n-2)$$
$$T(n) = 3^2(3T(n-3))$$
.
.
$$T(n) = 3^nT(n-n) = 3^nT(0) = 3^n$$

This clearly shows that the complexity of this function is $O(3^n)$.
Note: We can use the *Subtraction and Conquer* master theorem for this problem.

Problem-22 Find the complexity of the below recurrence:

$$T(n) = \begin{cases} 2T(n-1) - 1, if\ n > 0, \\ 1, \quad otherwise \end{cases}$$

Solution: Let us try solving this function with substitution.

$$T(n) = 2T(n-1) - 1$$
$$T(n) = 2(2T(n-2) - 1) - 1 = 2^2T(n-2) - 2 - 1$$
$$T(n) = 2^2(2T(n-3) - 2 - 1) - 1 = 2^3T(n-4) - 2^2 - 2^1 - 2^0$$
$$T(n) = 2^nT(n-n) - 2^{n-1} - 2^{n-2} - 2^{n-3} \ldots .2^2 - 2^1 - 2^0$$
$$T(n) = 2^n - 2^{n-1} - 2^{n-2} - 2^{n-3} \ldots .2^2 - 2^1 - 2^0$$
$$T(n) = 2^n - (2^n - 1)\ [note: 2^{n-1} + 2^{n-2} + \cdots + 2^0 = 2^n]$$
$$T(n) = 1$$

∴Complexity is O(1). Note that while the recurrence relation looks exponential the solution to the recurrence relation here gives a different result.

Problem-23 What is the running time of the following function?

```
void Function(int n) {
        int i=1, s=1;
        while( s <= n) {
                i++;
                s= s+i;
                printf("*");
        }
}
```

Solution: Consider the comments in below function:

```
void Function (int n) {
        int i=1, s=1;
        // s is increasing not at rate 1 but i
        while( s <= n) {
                i++;
                s= s+i;
                printf("*");
        }
}
```

We can define the terms 's' according to the relation $s_i = s_{i-1} + i$. The value of 'i' increases by one for each iteration. The value contained in 's' at the i^{th} iteration is the sum of the first 'i' positive integers. If k is the total number of iterations taken by the program, then *while* loop terminates if:

$$1 + 2 + \ldots + k = \frac{k(k+1)}{2} > n \Rightarrow k = O(\sqrt{n}).$$

Problem-24 Find the complexity of the function given below.

```
void Function(int n) {
        int i, count =0;
        for(i=1; i*i<=n; i++)
                count++;
}
```

Solution:

```
void Function(int n) {
        int i, count =0;
        for(i=1; i*i<=n; i++)
                count++;
}
```

In the above function the loop will end, if $i^2 \leq n \Rightarrow T(n) = O(\sqrt{n})$. The reasoning is same as that of Problem-23.

Problem-25 What is the complexity of the below program:

```
void function(int n) {
        int i, j, k , count =0;
        for(i=n/2; i<=n; i++)
                for(j=1; j + n/2<=n; j= j++)
                        for(k=1; k<=n; k= k * 2)
                                count++;
}
```

Solution: Consider the comments in the following function.

```
void function(int n) {
        int i, j, k , count =0;
        //outer loop execute n/2 times
        for(i=n/2; i<=n; i++)
                //Middle loop executes n/2 times
                for(j=1; j + n/2<=n; j= j++)
                        //outer loop execute logn times
                        for(k=1; k<=n; k= k * 2)
                                count++;
}
```

The complexity of the above function is $O(n^2 logn)$.

Problem-26 What is the complexity of the below program:

```
void function(int n) {
        int i, j, k , count =0;
        for(i=n/2; i<=n; i++)
                for(j=1; j<=n; j= 2 * j)
                        for(k=1; k<=n; k= k * 2)
                                count++;
}
```

Solution: Consider the comments in the following function.

```
void function(int n) {
        int i, j, k , count =0;
        //outer loop execute n/2 times
        for(i=n/2; i<=n; i++)
```

```
        //Middle loop executes logn times
        for(j=1; j<=n; j= 2 * j)
                //outer loop execute logn times
                for(k=1; k<=n; k= k*2)
                        count++;
}
```

The complexity of the above function is O($nlog^2n$).

Problem-27 Find the complexity of the below program.

```
function( int n ) {
        if(n == 1) return;
        for(int i = 1 ; i <= n ; i + + ) {
                for(int j= 1 ; j <= n ; j + + ) {
                        printf("*" );
                        break;
                }
        }
}
```

Solution: Consider the comments in the following function.

```
function( int n ) {
        //constant time
        if( n == 1 ) return;
        //outer loop execute n times
        for(int i = 1 ; i <= n ; i + + ) {
                // inner loop executes only time due to break statement.
                for(int j= 1 ; j <= n ; j + + ) {
                        printf("*" );
                        break;
                }
        }
}
```

The complexity of the above function is O(n). Even though the inner loop is bounded by n, but due to the break statement it is executing only once.

Problem-28 Write a recursive function for the running time $T(n)$ of the function given below. Prove using the iterative method that $T(n) = \Theta(n^3)$.

```
        function( int n ) {
                if( n == 1 ) return;
                for(int i = 1 ; i <= n ; i + + )
                        for(int j = 1 ; j <= n ; j + + )
                                printf("*" ) ;
                function( n-3 );
        }
```

Solution: Consider the comments in below function:

```
function (int n) {
        //constant time
        if( n == 1 ) return;
        //outer loop execute n times
        for(int i = 1 ; i <= n ; i + + )
                //inner loop executes n times
```

```
                for(int j = 1 ; j <= n ; j++ )
                        //constant time
                        printf("*" ) ;
        function( n-3 );
}
```

The recurrence for this code is clearly $T(n) = T(n-3) + cn^2$ for some constant $c > 0$ since each call prints out n^2 asterisks and calls itself recursively on n - 3. Using the iterative method we get: $T(n) = T(n-3) + cn^2$. Using the *Subtraction and Conquer* master theorem, we get $T(n) = \Theta(n^3)$.

Problem-29 Determine Θ bounds for the recurrence relation: $T(n) = 2T\left(\frac{n}{2}\right) + nlogn$.

Solution: Using Divide and Conquer master theorem, we get $O(nlog^2n)$.

Problem-30 Determine Θ bounds for the recurrence: $T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + T\left(\frac{n}{8}\right) + n$.

Solution: Substituting in the recurrence equation, we get: $T(n) \leq c1 * \frac{n}{2} + c2 * \frac{n}{4} + c3 * \frac{n}{8} + cn \leq k * n$, where k is a constant. This clearly says $\Theta(n)$.

Problem-31 Determine Θ bounds for the recurrence relation: $T(n) = T(\lceil n/2 \rceil) + 7$.

Solution: Using Master Theorem we get $\Theta(logn)$.

Problem-32 Prove that the running time of the code below is $\Omega(logn)$.

```
void Read(int n) {
        int k = 1;
        while( k < n )
                k = 3*k;
}
```

Solution: The *while* loop will terminate once the value of '*k*' is greater than or equal to the value of '*n*'. In each iteration the value of '*k*' is multiplied by 3. If i is the number of iterations, then '*k*' has the value of $3i$ after i iterations. The loop is terminated upon reaching i iterations when $3^i \geq n \leftrightarrow i \geq \log_3 n$, which shows that $i = \Omega\ (logn)$.

Problem-33 Solve the following recurrence.

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

Solution: By iteration:

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

...

$$T(n) = T(1) + \sum_{i=1}^{n} i(i-1)$$

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

$$T(n) = 1 + \frac{n((n+1)(2n+1)}{6} - \frac{n(n+1)}{2}$$

$$T(n) = \Theta(n^3)$$

Note: We can use the *Subtraction and Conquer* master theorem for this problem.

Problem-34 Consider the following program:

```
Fib[n]
if(n==0) then return 0
else if(n==1) then return 1
```

```
        else return Fib[n-1]+Fib[n-2]
```

Solution: The recurrence relation for running time of this program is: $T(n) = T(n-1) + T(n-2) + c$. Notice T($n$) has two recurrence calls indicating a binary tree. Each step recursively calls the program for n reduced by 1 and 2, so the depth of the recurrence tree is O(n). The number of leaves at depth n is 2^n since this is a full binary tree, and each leaf takes at least O(1) computation for the constant factor. Running time is clearly exponential in n and it is O(2^n).

Problem-35 Running time of following program?

```
function(n) {
        for(int i = 1; i <= n ; i + + )
                for(int j = 1 ; j <= n ; j+ = i )
                        printf(" * ") ;
}
```

Solution: Consider the comments in below function:

```
function (n) {
        //this loop executes n times
        for(int i = 1;  i <= n ; i + + )
                //this loop executes j times with j increase by the rate of i
                for(int j = 1 ; j <= n ; j+ = i )
                        printf( " * " ) ;
}
```

In the above code, inner loop executes n/i times for each value of i. Its running time is $n \times (\sum_{i=1}^{n} n/i) = O(nlogn)$.

Problem-36 What is the complexity of $\sum_{i=1}^{n} log\ i$?

Solution: Using the logarithmic property, $logxy = logx + logy$, we can see that this problem is equivalent to

$$\sum_{i=1}^{n} logi = log\,1 + log\,2 + \cdots + log\,n = log(1 \times 2 \times \ldots \times n) = log(n!) \ \le log(n^n) \le nlogn$$

This shows that that the time complexity = O($nlogn$).

Problem-37 What is the running time of the following recursive function (specified as a function of the input value n)? First write the recurrence formula and then find its complexity.

```
function(int n) {
        if(n <= 1) return;
        for (int i=1 ; i <= 3; i++ )
                f(⌈n/3⌉);
}
```

Solution: Consider the comments in below function:

```
function (int n) {
        //constant time
        if(n <= 1) return;
        //this loop executes with recursive loop of n/3 value
        for (int i=1 ; i <= 3; i++ )
                f(⌈n/3⌉);
}
```

We can assume that for asymptotical analysis $k = \lceil k \rceil$ for every integer $k \ge 1$. The recurrence for this code is $T(n) = 3T(\frac{n}{3}) + \Theta(1)$. Using master theorem, we get $T(n) = \Theta(n)$.

Problem-38 What is the running time of the following recursive function (specified as a function of the input value n)? First write a recurrence formula, and show its solution using induction.

```
function(int n) {
```

```
        if(n <= 1) return;
        for (int i=1 ; i <= 3 ; i++ )
            function (n – 1).
}
```

Solution: Consider the comments in below function:

```
function (int n) {
        //constant time
        if(n <= 1) return;
        //this loop executes 3 times with recursive call of n-1 value
        for (int i=1 ; i <= 3 ; i++ )
            function (n – 1).
}
```

The if statement requires constant time [O(1)]. With the for loop, we neglect the loop overhead and only count three times that the function is called recursively. This implies a time complexity recurrence:

$$T(n) = c, if\ n \leq 1;$$
$$= c + 3T(n-1), if\ n > 1.$$

Using the *Subtraction and Conquer* master theorem, we get $T(n) = \Theta(3^n)$.

Problem-39 Write a recursion formula for the running time $T(n)$ of the function whose code is below.

```
function (int n) {
        if(n <= 1) return;
        for(int i = 1; i < n; i + +)
            printf(" * ");
        function ( 0.8n ) ;
}
```

Solution: Consider the comments in below function:

```
function (int n) {
        if(n <= 1) return;          //constant time
        // this loop executes n times with constant time loop
        for(int i = 1; i < n; i + +)
            printf(" * ");
        //recursive call with 0.8n
        function ( 0.8n ) ;
}
```

The recurrence for this piece of code is $T(n) = T(.8n) + O(n) = T(4/5n) + O(n) = 4/5\ T(n) + O(n)$. Applying master theorem, we get $T(n) = O(n)$.

Problem-40 Find the complexity of the recurrence: $T(n) = 2T(\sqrt{n}) + logn$

Solution: The given recurrence is not in the master theorem form. Let us try to convert this to master theorem format by assuming $n = 2^m$. Applying logarithm on both sides gives, $logn = mlog2 \Rightarrow m = logn$. Now, the given function becomes,

$$T(n) = T(2^m) = 2T(\sqrt{2^m}) + m = 2T\left(2^{\frac{m}{2}}\right) + m.$$

To make it simple we assume $S(m) = T(2^m) \Rightarrow S(\frac{m}{2}) = T(2^{\frac{m}{2}}) \Rightarrow S(m) = 2S\left(\frac{m}{2}\right) + m$. Applying the master theorem would result $S(m) = O(mlogm)$. If we substitute $m = logn$ back, $T(n) = S(logn) = O((logn)\ loglogn)$.

Problem-41 Find the complexity of the recurrence: $T(n) = T(\sqrt{n}) + 1$

Solution: Applying the logic of Problem-40, gives $S(m) = S\left(\frac{m}{2}\right) + 1$. Applying the master theorem would result $S(m) = O(logm)$. Substituting $m = logn$, gives $T(n) = S(logn) = O(loglogn)$.

Problem-42 Find the complexity of the recurrence: $T(n) = 2T(\sqrt{n}) + 1$

Solution: Applying the logic of Problem-40, gives: $S(m) = 2S\left(\frac{m}{2}\right) + 1$. Using the master theorem results $S(m) = O(m^{log_2^2}) = O(m)$. Substituting $m = logn$ gives $T(n) = O(logn)$.

Problem-43 Find the complexity of the below function.

```
int Function (int n) {
        if(n <= 2) return 1;
        else return (Function (floor(sqrt(n))) + 1);
}
```

Solution: Consider the comments in below function:

```
int Function (int n) {
        //constant time
        if(n <= 2) return 1;
        else    // executes √n + 1 times
                return (Function (floor(sqrt(n))) + 1);
}
```

For the above code, the recurrence function can be given as: $T(n) = T(\sqrt{n}) + 1$. This is same as that of Problem-41.

Problem-44 Analyze the running time of the following recursive psuedocode as a function of n.

```
void function(int n) {
        if( n < 2 ) return;
        else counter = 0;
        for i = 1 to 8 do
                function (n/2);
        for i =1 to n^3 do
                counter = counter + 1;
}
```

Solution: Consider the comments in below psuedocode and call running time of function(n) as $T(n)$.

```
void function(int n) {
        if( n < 2 ) return;   //constant time
        else    counter = 0;
        // this loop executes 8 times with n value half in every call
        for i = 1 to 8 do
                function (n/2);
        // this loop executes n^3 times with constant time loop
        for i =1 to n^3 do
                counter = counter + 1;
}
```

$T(n)$ can be defined as follows:

$$T(n) = 1 \; if \; n < 2,$$
$$= 8T(\frac{n}{2}) + n^3 + 1 \; otherwise.$$

Using the master theorem gives, $T(n) = \Theta(n^{log_2^8} logn) = \Theta(n^3 logn)$.

Problem-45 Find the complexity of the below psuedocode.

```
temp = 1
repeat
        for i = 1 to n
                temp = temp + 1;
```