CHAPTER

1

INTRODUCTION

The objective of this chapter is to explain the importance of the analysis of algorithms, their notations, relationships and solving as many problems as possible. Let us first focus on understanding the basic elements of algorithms, the importance of algorithm analysis, and then slowly move toward the other topics as mentioned above. After completing 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 this equation. The important thing that we need to understand is that the equation has names (x and y), which hold values (data). That means the *names* (x and y) are placeholders for representing data. Similarly, in computer science programming we need something for holding data, and *variables* is the way to do that.

1.2 Data Types

In the above-mentioned equation, the variables x and y can take any values such as integral numbers (10, 20), real numbers (0.23, 5.5), or just 0 and 1. To solve the equation, we need to relate them to the kind of values they can take, and *data type* is the name used in computer science programming for this purpose. A *data type* in a programming language is a set of data with predefined values. 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 we want to code it, it's very difficult to provide the solution in terms of zeros and ones. To help users, programming languages and compilers provide us with 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 an *integer*. Similarly, combining 4 bytes (32 bits) and calling it a *float*. A data type reduces the coding effort. 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 that are defined by system are called *primitive* data types. The primitive data types 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, the compiler and the 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 change.

For example, "*int*" may take 2 bytes or 4 bytes. If it takes 2 bytes (16 bits), then the total possible values are minus 32,768 to plus 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 and +2,147,483,647 (-2^{31} *to* $2^{31}-1$). The same is the case with other data types.

User defined data types

If the system-defined data types are not enough, then most programming languages allow the users to define their own data types, called *user − defined data types*. Good examples of user defined data types are: structures in *C/C++* and classes in *Java*. For example, in the snippet below, we are combining many system-defined data types and calling the user defined data type by the 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 Structures

Based on the discussion above, 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. A *data structure* is a special 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. *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, etc.) support basic operations such as addition and subtraction. The system provides the implementations for the primitive data types. For user-defined data types we also 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 problems, we combine the data structures with their operations and we call this *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 others. 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 worry about the implementation details. They come into the 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 *omelette*. To prepare an omelette, we follow the steps given below:

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 omelette), we are providing a step-by-step procedure for solving it. The formal definition of an algorithm can be stated 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 the 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 bicycle. Depending on the availability and convenience, we choose the one that suits us. Similarly, in computer science, multiple algorithms are available for solving the same problem (for example, a sorting problem has many algorithms, like insertion sort, selection sort, quick sort and many more). Algorithm analysis helps us to determine which algorithm is most efficient in terms of time and space consumed.

1.7 Goal of the Analysis of Algorithms

The goal of the *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, developer 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 the number of elements in the input, and depending on the problem type, the input may be of different types. The following are the common types of inputs.

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

1.9 How to Compare Algorithms

To compare algorithms, let us define a 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 express the running time of a 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 go to a shop to buy a car and a bicycle. If your friend sees you there and asks what you are buying, then in general you say *buying a car*. This is because the cost of the car is high compared to the cost of the bicycle (approximating the cost of the bicycle to the cost of the car).

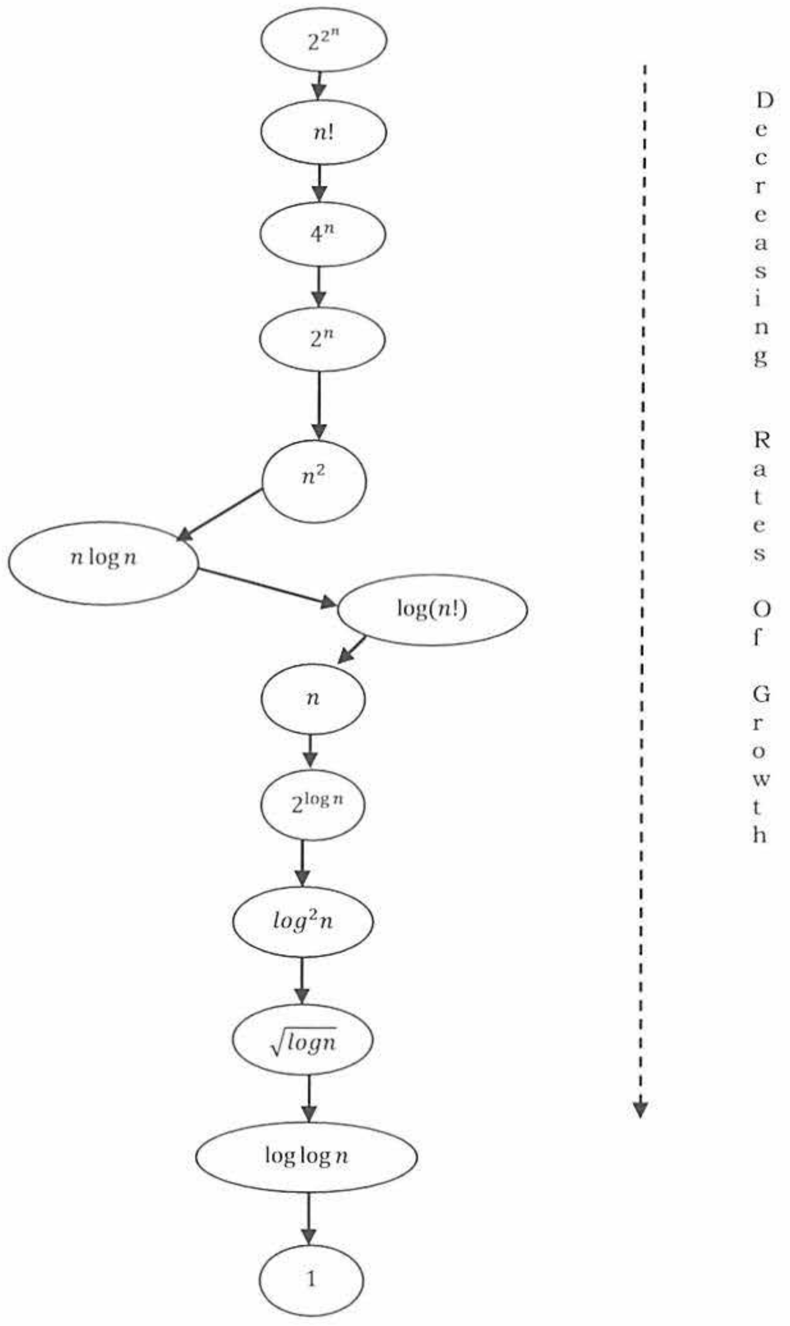
$$Total\ Cost = cost\_of\_car + cost\_of\_bicycle$$
$$Total\ Cost \approx cost\_of\_car\ (approximation)$$

For the above-mentioned example, we can represent the cost of the car and the cost of the bicycle 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 case below, n^4, $2n^2$, $100n$ and 500 are the individual costs of some function and approximate 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 Rates of Growth

The diagram below shows the relationship between different rates of growth.



Below is the list of growth rates you will come across in the following 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

1.12 Types of Analysis

To analyze the given algorithm, we need to know with which inputs the algorithm takes less time (performing well) and with which inputs the algorithm takes a long 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 the case where it takes less time and another for the case where it takes more time.

In general, the first case is called the *best case* and the second case is called the *worst case* for 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 a long time.
  - Input is the one for which the algorithm runs the slowest.
- **Best case**
  - Defines the input for which the algorithm takes the least 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 the best, worst and average cases in the form of expressions. As an example, let $f(n)$ be the function which represents the given algorithm.

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

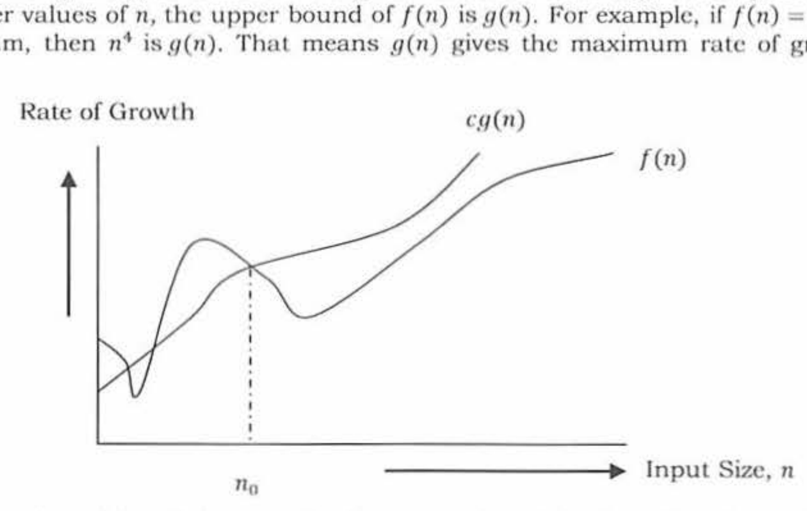
Similarly for the average case. The expression defines the inputs with which the algorithm takes the average running time (or memory).

1.13 Asymptotic Notation

Having the expressions for the best, average and worst cases, for all three cases we need to identify the upper and lower bounds. To represent these upper and lower bounds, we need some kind of syntax, and that is the subject of the 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.



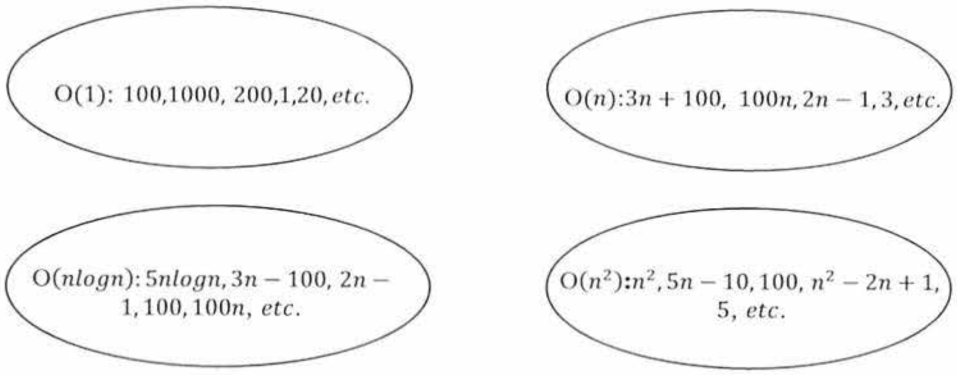
Let us see the O–notation with a little more detail. O–notation defined as $O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0\}$. $g(n)$ is an asymptotic tight upper bound for $f(n)$. Our objective is to give the smallest rate of growth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

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

Big-O Visualization

$O(g(n))$ is the set of functions with smaller or the 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 about the rate of growth.


O(1): 100,1000, 200,1,20, etc.
O(n):3n + 100, 100n, 2n − 1, 3, etc.
O(nlogn): 5nlogn, 3n − 100, 2n − 1, 100, 100n, etc.
O(n²):n², 5n − 10, 100, n² − 2n + 1, 5, etc.

Big-O Examples

Example-1 Find upper bound for $f(n) = 3n + 8$

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

Example-6 Find upper bound for $f(n) = 410$

Solution: $410 \leq 410$, for all $n \geq 1$
$\therefore 410 = O(1)$ with $c = 1$ and $n_0 = 1$

No Uniqueness?

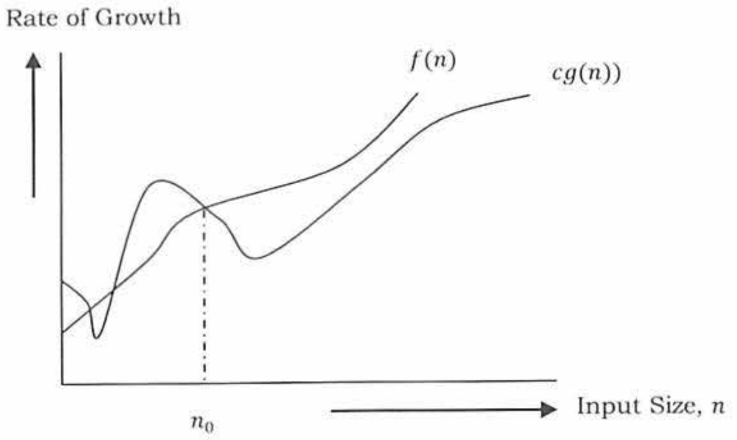
There is no unique set of values for n_0 and c in proving the asymptotic bounds. Let us consider, $100n + 5 = O(n)$. For this function there are multiple n_0 and c values possible.

Solution1: $100n + 5 \leq 100n + n = 101n \leq 101n$, for all $n \geq 5$, $n_0 = 5$ and $c = 101$ is a solution.

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

1.15 Omega-Ω Notation

Similar to the 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 the largest rate of growth $g(n)$ which is less than or equal to the given algorithm's rate of growth $f(n)$.

Ω Examples

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

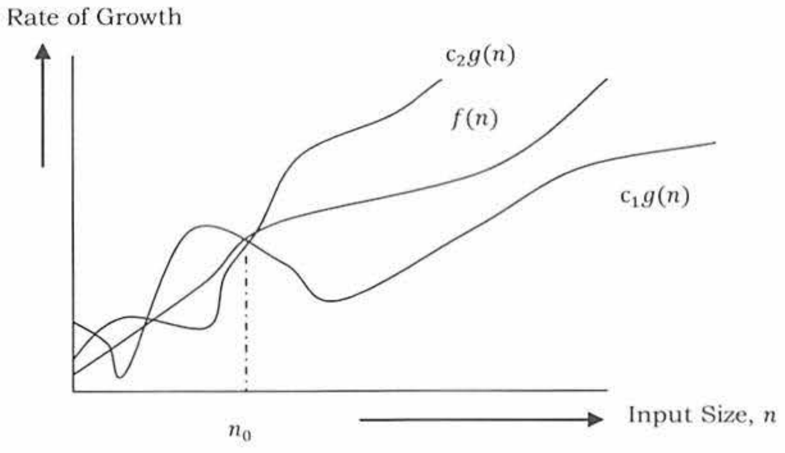
Solution: $\exists\, c,\ n_0$ Such that: $0 \leq cn^2 \leq 5n^2 \Rightarrow cn^2 \leq 5n^2 \Rightarrow c = 1$ and $n_0 = 1$
$\therefore 5n^2 = \Omega(n^2)$ with $c = 1$ and $n_0 = 1$

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

1.16 Theta-Θ Notation



This notation decides whether the upper and lower bounds of a given function (algorithm) are the same. The average running time of an algorithm is always between the lower bound and the upper bound. If the upper bound (O) and lower bound (Ω) give the same result, then the 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 the best case is $g(n) = O(n)$.

In this case, the rates of growth in the best case and worst case are the same. As a result, the average case will also be the same. For a given function (algorithm), if the rates of growth (bounds) for O and Ω are not the same, then the rate of growth for the Θ case may not be the same. In this case, we need to consider all possible time complexities and take the average of those (for example, for a quick sort average case, refer to the *Sorting* chapter).

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 $0 \le c_1 g(n) \le f(n) \le c_2 g(n)$ for all $n \ge n_0\}$. $g(n)$ 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: $\frac{n^2}{5} \le \frac{n^2}{2} - \frac{n}{2} \le n^2$, for all, $n \ge 1$

$\therefore \frac{n^2}{2} - \frac{n}{2} = \Theta(n^2)$ with $c_1 = 1/5, c_2 = 1$ and $n_0 = 1$

Example 2 Prove $n \neq \Theta(n^2)$

Solution: $c_1 n^2 \le n \le c_2 n^2 \Rightarrow$ only holds for: $n \le 1/c_1$

$\therefore n \neq \Theta(n^2)$

Example 3 Prove $6n^3 \neq \Theta(n^2)$

Solution: $c_1 n^2 \le 6n^3 \le c_2 n^2 \Rightarrow$ only holds for: $n \le c_2/6$

$\therefore 6n^3 \neq \Theta(n^2)$

Example 4 Prove $n \neq \Theta(logn)$

Solution: $c_1 logn \le n \le c_2 logn \Rightarrow c_2 \ge \frac{n}{\log n}, \forall\, n \ge n_0$ – Impossible

Important Notes

For analysis (best case, worst case and average), we try to give the 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 the upper bound (O) and lower bound (Ω) and average running time (Θ) may not always be possible. For example, if we are discussing the best case of an algorithm, we try to give the upper bound (O) and lower bound (Ω) and average running time (Θ).

In the remaining chapters, we generally focus on the upper bound (O) because knowing the lower bound (Ω) of an algorithm is of no practical importance, and we use the Θ notation if the upper bound (O) and lower bound (Ω) are the same.

1.17 Why is it called Asymptotic Analysis?

From the discussion above (for all 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 another 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 a curve an *asymptotic curve*. In other terms, $g(n)$ is the asymptotic curve for $f(n)$. For this reason, we call algorithm analysis *asymptotic analysis*.

1.18 Guidelines for Asymptotic Analysis

There are some general rules to help us determine 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 in range(0,n):
    print 'Current Number :', i   #constant time
```

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

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

```
# outer loop executed n times
for i in range(0,n):
    # inner loop executes n times
    for j in range(0,n):
        print 'i value %d and j value %d' % (i,j) #constant time
```

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

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

```
n = 100
# executes n times
for i in range(0,n):
    print 'Current Number :', i                          #constant time
# outer loop executed n times
for i in range(0,n):
    # inner loop executes n times
    for j in range(0,n):
        print 'i value %d and j value %d' % (i,j)        #constant time
```

Total time = $c_0 + c_1n + c_2n^2 = O(n^2)$.

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

```
if n == 1:                          #constant time
   print "Wrong Value"
   print n
else:
   for i in range(0,n):             #n times
     print 'Current Number :', i    #constant time
```

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

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:

```
def Logarithms(n):
  i = 1
  while i <= n:
          i= i * 2
          print i
Logarithms(100)
```

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 $2^k = n$ and we come out of loop. Taking logarithm on both sides, gives

$log(2^k) = logn$
$klog2 = logn$
$k = logn$ //if we assume base-2

Total time = $O(logn)$.

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

```
def Logarithms(n):
  i = n
  while i >= 1:
          i= i // 2
          print i
Logarithms(100)
```

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

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

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

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 (Also discussed in detail in the *Divide and Conquer* chapter) divide the problem into sub-problems, each of which is part of the original problem, and then perform some additional work to compute the final answer. As an example, a merge sort algorithm [for details, refer to *Sorting* chapter] operates on two sub-problems, 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 $T(n) = \Theta\left(n^{log_b^a}\right)$
2) If $a = b^k$
    a. If $p > -1$, then $T(n) = \Theta\left(n^{log_b^a} log^{p+1} n\right)$
    b. If $p = -1$, then $T(n) = \Theta\left(n^{log_b^a} loglogn\right)$
    c. If $p < -1$, then $T(n) = \Theta\left(n^{log_b^a}\right)$
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 Divide and Conquer Master Theorem: Problems & Solutions

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: $T(n) = 2^n T(n/2) + n^n \Rightarrow$ 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: $T(n) = 2T(n/4) + n^{0.51} \Rightarrow T\,(n) = \Theta(n^{0.51})$ (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

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

for some constants $c, a > 0, b > 0, k \ge 0$, 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 Method of Guessing and Confirming

Now, let us discuss a method which can be used to solve any recurrence. The basic idea behind this method is:

guess the answer; and then *prove* it correct by induction.

In other words, it addresses the question: What if the given recurrence doesn't seem to match with any of these (master theorem) methods? If we guess a solution and then try to verify our guess inductively, usually either the proof will succeed (in which case we are done), or the proof will fail (in which case the failure will help us refine our guess).

As an example, consider the recurrence $T(n) = \sqrt{n}\, T(\sqrt{n}) + n$. This doesn't fit into the form required by the Master Theorems. Carefully observing the recurrence gives us the impression that it is similar to the divide and conquer method (dividing the problem into $\sqrt{n}$ subproblems each with size $\sqrt{n}$). As we can see, the size of the subproblems at the first level of recursion is n. So, let us guess that $T(n) = O(nlogn)$, and then try to prove that our guess is correct.

Let's start by trying to prove an *upper* bound $T(n) \le cnlogn$:

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\le \sqrt{n}.\, c\sqrt{n}\, log\sqrt{n} + n \\ &= n.\ c\ log\sqrt{n} + n \\ &= n.c.\tfrac{1}{2}.\, logn + n \\ &\le cnlogn \end{aligned}$$

The last inequality assumes only that $1 \le c.\frac{1}{2}.logn$. This is correct if n is sufficiently large and for any constant c, no matter how small. From the above proof, we can see that our guess is correct for the upper bound. Now, let us prove the *lower* bound for this recurrence.

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\ge \sqrt{n}.\, k\, \sqrt{n}\, log\sqrt{n} + n \\ &= n.\ k\ log\sqrt{n} + n \\ &= n.k.\tfrac{1}{2}.\, logn + n \\ &\ge knlogn \end{aligned}$$

The last inequality assumes only that $1 \ge k.\frac{1}{2}.logn$. This is incorrect if n is sufficiently large and for any constant k. From the above proof, we can see that our guess is incorrect for the lower bound.

From the above discussion, we understood that $\Theta(nlogn)$ is too big. How about $\Theta(n)$? The lower bound is easy to prove directly:

$$T(n) = \sqrt{n}\, T(\sqrt{n}) + n \ \ge n$$

Now, let us prove the upper bound for this $\Theta(n)$.

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\le \sqrt{n}.c.\sqrt{n} + n \\ &= n.\ c + n \\ &= n\ (c+1) \\ &\nleq cn \end{aligned}$$

From the above induction, we understood that $\Theta(n)$ is too small and $\Theta(nlogn)$ is too big. So, we need something bigger than n and smaller than $nlogn$. How about $n\sqrt{logn}$?

Proving the upper bound for $n\sqrt{logn}$:

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\leq \sqrt{n}.c.\sqrt{n}\sqrt{log\sqrt{n}} + n \\ &= n.\ c.\tfrac{1}{\sqrt{2}}\ log\sqrt{n} + n \\ &\leq cnlog\sqrt{n} \end{aligned}$$

Proving the lower bound for $n\sqrt{logn}$:

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\geq \sqrt{n}.k.\sqrt{n}\sqrt{log\sqrt{n}} + n \\ &= n.\ k.\tfrac{1}{\sqrt{2}}\ log\sqrt{n} + n \\ &\ngeq knlog\sqrt{n} \end{aligned}$$

The last step doesn't work. So, $\Theta(n\sqrt{logn})$ doesn't work. What else is between n and $nlogn$? How about $nloglogn$?

Proving upper bound for $nloglogn$:

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\leq \sqrt{n}.c.\sqrt{n}loglog\sqrt{n} + n \\ &= n.\ c.loglogn\text{-}c.n + n \\ &\leq cnloglogn, \text{ if } c \geq 1 \end{aligned}$$

Proving lower bound for $nloglogn$:

$$\begin{aligned} T(n) &= \sqrt{n}\, T(\sqrt{n}) + n \\ &\geq \sqrt{n}.k.\sqrt{n}loglog\sqrt{n} + n \\ &= n.\ k.loglogn\text{-}k.n + n \\ &\geq knloglogn, \text{ if } k \leq 1 \end{aligned}$$

From the above proofs, we can see that $T(n) \leq cnloglogn$, if $c \geq 1$ and $T(n) \geq knloglogn$, if $k \leq 1$. Technically, we're still missing the base cases in both proofs, but we can be fairly confident at this point that $T(n) = \Theta(nloglogn)$.

1.26 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 *change 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 the total of the real costs for the sequence. This artificial cost is called the amortized cost of an operation. 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 the next few operations become easier.

Example: Let us consider an array of elements from which we want to find the 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. The cost of performing the 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.

1.27 Algorithms Analysis: Problems & Solutions

Note: From the following problems, try to understand the cases which have 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.

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

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.

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

∴ Time 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?

```
def Function(n):
        i = s = 1
        while  s < n:
                i = i+1
                s = s+i
                print("*")
Function(20)
```

Solution: Consider the comments in the below function:

```
def Function(n):
        i = s = 1
        while  s < n:                          # s is increasing not at rate 1 but i
                i = i+1
                s = s+i
                print("*")
Function(20)
```

We can define the '*s*' terms according to the relation $s_i = s_{i-1} + i$. The value of '*i*' increases by 1 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 the *while* loop terminates if:

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

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

```
def Function(n):
        i = 1
        count = 0
        while i*i <n:
                count = count +1
                i = i + 1
        print(count)
Function(20)
```

Solution: In the above-mentioned function the loop will end, if $i^2 \leq n \Rightarrow T(n) = O(\sqrt{n})$. This is similar to Problem-23.

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

```
def Function(n):
        count = 0
        for i in range(n/2, n):
                j = 1
                while j + n/2 <= n:
                        k = 1
                        while k <= n:
                                count = count + 1
                                k = k * 2
                        j = j + 1
        print (count)
Function(20)
```

Solution: Observe the comments in the following function.

```
def Function(n):
        count = 0
        for i in range(n/2, n):                 #Outer loop execute n/2 times
                j = 1
                while j + n/2 <= n:             #Middle loop executes n/2 times
                        k = 1
                        while k <= n:           #Inner loop execute logn times
                                count = count + 1
                                k = k * 2
                        j = j + 1
        print (count)
Function(20)
```

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

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

```
def Function(n):
        count = 0
        for i in range(n/2, n):
                j = 1
                while j + n/2 <= n:
                        k = 1
                        while k <= n:
                                count = count + 1
                                k = k * 2
                        j = j * 2
        print (count)
Function(20)
```

Solution: Consider the comments in the following function.

```
def Function(n):
        count = 0
        for i in range(n/2, n):                 #Outer loop execute n/2 times
                j = 1
                while j + n/2 <= n:             #Middle loop executes logn times
                k = 1
                while k <= n:                    #Inner loop execute logn times
                        count = count + 1
                        k = k * 2
                j = j * 2
        print (count)
    Function(20)
```

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

Problem-27 Find the complexity of the program below.

```
def Function(n):
        count = 0
        for i in range(n/2, n):
                j = 1
                while j + n/2 <= n:
                        break
                        j = j * 2
        print (count)
Function(20)
```

Solution: Consider the comments in the function below.

```
def Function(n):
        count = 0
        for i in range(n/2, n):                  #Outer loop execute n/2 times
                j = 1
                while j + n/2 <= n:              #Middle loop has break statement
                        break
                        j = j * 2
        print (count)
Function(20)
```

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

```
def Function(n):
        count = 0
        if n <= 0:
                return
        for i in range(0, n):
                for j in range(0, n):
                        count = count + 1
        Function(n-3)
        print (count)
Function(20)
```

Solution: Consider the comments in the function below:

```
def Function(n):
        count = 0
        if n <= 0:
                return
        for i in range(0, n):                    #Outer loop executes n times
                for j in range(0, n):            #Outer loop executes n times
                        count = count + 1
        Function(n-3)                            #Recursive call
        print (count)
Function(20)
```

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: $T(n) \le c1 * \frac{n}{2} + c2 * \frac{n}{4} + c3 * \frac{n}{8} + cn \le k * n$, 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 $\Omega(logn)$.

```
def Read(n):
        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 $3^i \ge n \leftrightarrow i \ge \log_3 n$, which shows that $i = \Omega(logn)$.

Problem-33 Solve the following recurrence.

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

Solution: By iteration:

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

$$\ldots$$

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

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

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

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

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

Problem-34 Consider the following program:

```
def Fib(n):
        if n == 0: return 0
        elif n == 1: return 1
        else: return Fib(n-1)+ Fib(n-2)
print(Fib(3))
```

Solution: The recurrence relation for the running time of this program is: $T(n) = T(n-1) + T(n-2) + c$. Note 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) computations for the constant factor. Running time is clearly exponential in n and it is O(2^n).

Problem-35 Running time of following program?

```
def Function(n):
        count = 0
        if n <= 0:
                return
        for i in range(0, n):
                j = 1
                while j <n:
                        j = j + i
                        count = count + 1
        print (count)
Function(20)
```

Solution: Consider the comments in the function below:

```
def Function(n):
    count = 0
    if n <= 0:
        return
    for i in range(0, n):          #Outer loop executes n times
        j = 1                      #Inner loop executes j increase by the rate of i
        while j <n:
            j = j + i
            count = count + 1
    print (count)
Function(20)
```

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

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

This shows 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.

```
def Function(n):
    if n <= 0:
        return
    for i in range(0, 3):
        Function(n/3)
Function(20)
```

Solution: Consider the comments in the below function:

```
def Function(n):
    if n <= 0:
        return
    for i in range(0, 3):     #This loop executes 3 times with recursive value of n/3 value
        Function(n/3)
Function(20)
```

We can assume that for asymptotical analysis $k = \lceil k \rceil$ for every integer $k \geq 1$. 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.

```
def Function(n):
    if n <= 0:
        return
    for i in range(0, 3):     #This loop executes 3 times with recursive value of n/3 value
        Function(n-1)
Function(20)
```

Solution: Consider the comments in the function below:

```
def Function(n):
    if n <= 0:
        return
    for i in range(0, 3):     #This loop executes 3 times with recursive value of n – 1 value
        Function(n-1)
Function(20)
```

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.

```
def Function3(n):
        if n <= 0:
                return
        for i in range(0, 3):        #This loop executes 3 times with recursive value of n/3  value
                Function3(0.8 * n)
Function3(20)
```

Solution: Consider the comments in the function below:

```
def Function3(n):
        if n <= 0:
                return
        for i in range(0, 3):        #This loop executes 3 times with recursive value of 0.8n value
                Function3(0.8 * n)
Function3(20)
```

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 format. Let us try to convert this to the master theorem format by assuming $n = 2^m$. Applying the logarithm on both sides gives, $logn = mlog2 \Longrightarrow m = logn$. 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 $S(m) = T(2^m) \Longrightarrow S(\frac{m}{2}) = T(2^{\frac{m}{2}}) \Longrightarrow S(m) = 2S\left(\frac{m}{2}\right) + m.$
Applying the master theorem format would result in $S(m) = O(mlogm)$.
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 in $S(m) = O(logm)$. 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 $S(m) = O\left(m^{log_2^2}\right) = O(m)$. Substituting $m = logn$ gives $T(n) = O(logn)$.

Problem-43 Find the complexity of the below function.

```
import math
count = 0
def Function(n):
        global count
        if n <= 2:
                return 1
        else:
                Function(round(math.sqrt(n)))
                count = count + 1
                return count
print(Function(200))
```

Solution: Consider the comments in the function below:

```
import math
count = 0
def Function(n):
        global count
        if n <= 2:
                return 1
        else:
                Function(round(math.sqrt(n)))    #Recursive call with √n value
                count = count + 1
                return count
print(Function(200))
```

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 pseudo-code as a function of n.

```
def function(n):
        if (n < 2):
                return
        else:
                counter = 0
        for i in range(0,8):
                function (n/2)
        for i in range(0,n**3):
                counter = counter + 1
```

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

```
def function(n):
    if (n < 2):                                    # Constant time
        return
    else:
            counter = 0                    # Constant time
    for i in range(0,8):                   # This loop executes 8 times with n value half in every call
            function (n/2)
    for i in range(0,n**3):                # This loop executes $n^3$times with constant time loop
                    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: $T(n) = \Theta(n^{log_2^n} logn) = \Theta(n^3 logn)$.

Problem-45 Find the complexity of the below pseudocode.

```
count = 0
def Function(n):
        global count
        count  = 1
        if n <= 0:
                return
        for i in range(0, n):
                count =  count + 1
        n  = n//2;
        Function(n)
        print count

Function(200)
```

Solution: Consider the comments in the pseudocode below:

```
count = 0
def Function(n):
    global count
    count  = 1
    if n <= 0:
            return
    for i in range(1, n):        # This loops executes n times
            count =  count + 1
    n  = n//2;          #Integer Divison
    Function(n)         #Recursive call with $\frac{n}{2}$ value
    print count
Function(200)
```

The recurrence for this function is $T(n) = T(n/2) + n$. Using master theorem we get $T(n) = O(n)$.

Problem-46 Running time of the following program?

```
def Function(n):
        for i in range(1, n):
                j = 1
                while j <= n:
                        j = j * 2
                        print("*")
    Function(20)
```

Solution: Consider the comments in the below function:

```
def Function(n):
    for i in range(1, n):          # This loops executes n times
        j = 1
        while j <= n:              # This loops executes logn times from our logarithms guideline
            j = j * 2
            print("*")
Function(20)
```

Complexity of above program is: $O(nlogn)$.

Problem-47 Running time of the following program?

```
    def Function(n):
            for i in range(0, n/3):
                    j = 1
                    while j <= n:
                            j = j + 4
                            print("*")
    Function(20)
```

Solution: Consider the comments in the below function:

```
def Function(n):
    for i in range(0, n/3):        #This loops executes n/3 times
        j = 1
        while j <= n:              #This loops executes n/4 times
            j = j + 4
            print("*")
Function(20)
```

The time complexity of this program is: $O(n^2)$.

Problem-48 Find the complexity of the below function:

```
  def Function(n):
     if n <= 0:
             return
     print ("*")
     Function(n/2)
     Function(n/2)
     print ("*")
  Function(20)
```

Solution: Consider the comments in the below function:

```
def Function(n):
    if n <= 0:          #Constant time
        return
    print ("*")         #Constant time
    Function(n/2)       #Recursion with n/2 value
    Function(n/2)       #Recursion with n/2 value
    print ("*")
Function(20)
```

The recurrence for this function is: $T(n) = 2T\left(\frac{n}{2}\right) + 1$. Using master theorem, we get $T(n) = O(n)$.

Problem-49 Find the complexity of the below function:

```
    count = 0
    def Logarithms(n):
            i = 1
            global count
            while i <= n:
                j = n
                while j > 0:
                        j = j//2
                        count = count + 1
                i= i * 2
        return count
print(Logarithms(10))
```

Solution:

```
count = 0
def Logarithms(n):
    i = 1
    global count
    while i <= n:
        j = n
        while j > 0:
            j = j//2 # This loops executes logn times from our logarithms guideline
            count = count + 1
        i= i * 2          # This loops executes logn times from our logarithms guideline
    return count

print(Logarithms(10))
```

Time Complexity: $O(logn * logn) = O(log^2n)$.

Problem-50 $\sum_{1 \le k \le n} O(n)$, where O($n$) stands for order n is:
(a) O(n) (b) O(n^2) (c) O(n^3) (d) O($3n^2$) (e) O($1.5n^2$)

Solution: (b). $\sum_{1 \le k \le n} O(n) = O(n) \sum_{1 \le k \le n} 1 = O(n^2)$.

Problem-51 Which of the following three claims are correct?
I $(n + k)^m = \Theta(n^m)$, where k and m are constants II $2^{n+1} = O(2^n)$ III $2^{2n+1} = O(2^n)$
(a) I and II (b) I and III (c) II and III (d) I, II and III

Solution: (a). (I) $(n + k)^m = n^k + c1*n^{k-1} + ... k^m = \Theta(n^k)$ and (II) $2^{n+1} = 2*2^n = O(2^n)$

Problem-52 Consider the following functions:
f(n) = 2^n g(n) = $n!$ h(n) = n^{logn}
Which of the following statements about the asymptotic behavior of f(n), g(n), and h(n) is true?
(A) f(n) = O(g(n)); g(n) = O(h(n)) (B) f(n) = Ω (g(n)); g(n) = O(h(n))
(C) g(n) = O(f(n)); h(n) = O(f(n)) (D) h(n) = O(f(n)); g(n) = Ω (f(n))

Solution: **(D)**. According to the rate of growth: h(n) < f(n) < g(n) (g(n) is asymptotically greater than f(n), and f(n) is asymptotically greater than h(n)). We can easily see the above order by taking logarithms of the given 3 functions: $lognlogn < n < log(n!)$. Note that, $log(n!) = O(nlogn)$.

Problem-53 Consider the following segment of C-code:

```
j = 1
while j <=n:
    j = j*2
```

The number of comparisons made in the execution of the loop for any $n > 0$ is:
(A) ceil(log_2^n)+ 1 (B) n (C) ceil(log_2^n) (D) floor(log_2^n) + 1

Solution: (a). Let us assume that the loop executes k times. After k^{th} step the value of j is 2^k. Taking logarithms on both sides gives $k = log_2^n$. Since we are doing one more comparison for exiting from the loop, the answer is ceil(log_2^n)+ 1.

Problem-54 Consider the following C code segment. Let T(n) denote the number of times the for loop is executed by the program on input n. Which of the following is true?

```
import math
def IsPrime(n):
    for i in range(2, math.sqrt(n)):
        if n%i == 0:
            print("Not Prime")
            return 0
    return 1
```

(A) T(n) = O($\sqrt{n}$) and T(n) = Ω($\sqrt{n}$) (B) T(n) = O($\sqrt{n}$) and T(n) = Ω(1)
(C) T(n) = O(n) and T(n) = Ω($\sqrt{n}$) (D) None of the above

Solution: (B). Big O notation describes the tight upper bound and Big Omega notation describes the tight lower bound for an algorithm. The *for* loop in the question is run maximum $\sqrt{n}$ times and minimum 1 time. Therefore, $T(n) = O(\sqrt{n})$ and $T(n) = \Omega(1)$.

Problem-55 In the following C function, let $n \geq m$. How many recursive calls are made by this function?

```
def gcd(n,m):
        if n%m ==0:
            return m
        n = n%m
        return gcd(m,n)
```

(A) $\Theta(log_2^n)$ (B) $\Omega(n)$ (C) $\Theta(log_2 log_2^n)$ (D) $\Theta(n)$

Solution: No option is correct. Big O notation describes the tight upper bound and Big Omega notation describes the tight lower bound for an algorithm. For $m = 2$ and for all $n = 2^i$, the running time is O(1) which contradicts every option.

Problem-56 Suppose $T(n) = 2T(n/2) + n$, T(0)=T(1)=1. Which one of the following is false?

(A) $T(n) = O(n^2)$ (B) $T(n) = \Theta(nlogn)$ (C) $T(n) = \Omega(n^2)$ (D) $T(n) = O(nlogn)$

Solution: (C). Big O notation describes the tight upper bound and Big Omega notation describes the tight lower bound for an algorithm. Based on master theorem, we get $T(n) = \Theta(nlogn)$. This indicates that tight lower bound and tight upper bound are the same. That means, $O(nlogn)$ and $\Omega(nlogn)$ are correct for given recurrence. So option (C) is wrong.

Problem-57 Find the complexity of the below function:

```
def Function(n):
   for i in range(1, n):
            j = i
            while j <i*i:
                    j = j + 1
                    if j %i == 0:
                            for k in range(0, j):
                                    print(" * ")
Function(10)
```

Solution:

```
def Function(n):
      for i in range(1, n):                         # Executes n times
            j = i
            while j <i*i:                           # Executes n*n times
                    j = j + 1
                    if j %i == 0:
                            for k in range(0, j):   #Executes j times = (n*n) times
                                    print(" * ")
Function(10)
```

Time Complexity: $O(n^5)$.

Problem-58 To calculate 9^n, give an algorithm and discuss its complexity.

Solution: Start with 1 and multiply by 9 until reaching 9^n.

Time Complexity: There are $n - 1$ multiplications and each takes constant time giving a $\Theta(n)$ algorithm.

Problem-59 For Problem-58, can we improve the time complexity?

Solution: Refer to the *Divide and Conquer* chapter.

Problem-60 Find the complexity of the below function:

```
def Function(n):
      sum = 0
      for i in range(0, n-1):
            if i > j:
                    sum = sum + 1
            else:
                    for k in range(0, j):
                            sum = sum - 1
```

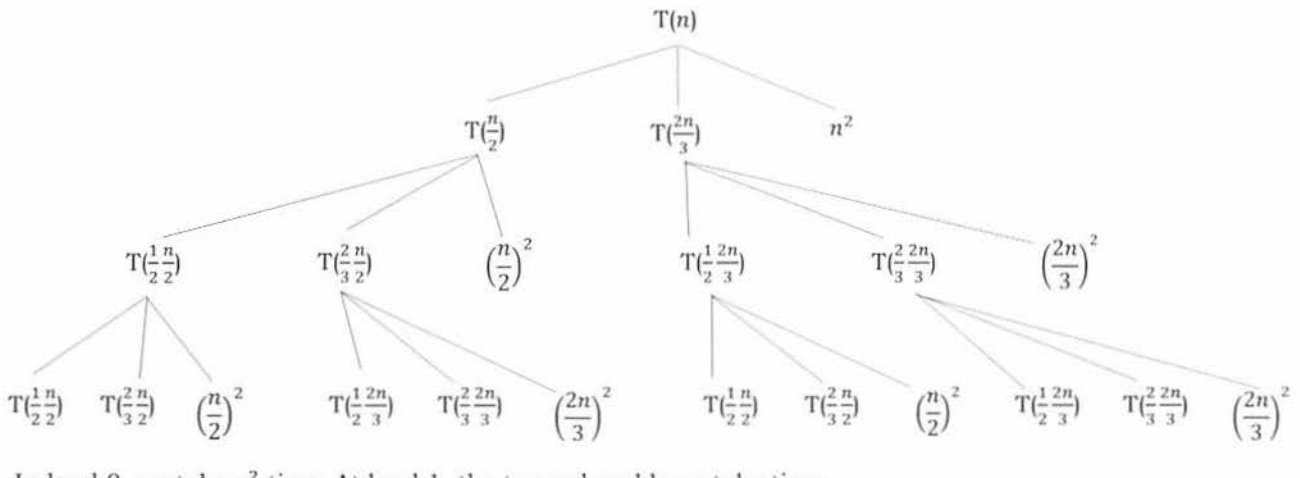
```
        print (sum)
    Function(10)
```

Solution: Consider the *worst − case* and we can ignore the value of j.

```
def Function(n):
    sum = 0
    for i in range(0, n-1):                    # Executes n times
            if i > j:
                    sum = sum + 1              # Executes n times
            else:
                    for k in range(0, j):      # Executes n times
                            sum = sum - 1
    print (sum)
Function(10)
```

Time Complexity: $O(n^2)$.

Problem-61 Solve the following recurrence relation using the recursion tree method: $T(n)=T(\frac{n}{2}) + T(\frac{2n}{3}) + n^2$.
Solution: How much work do we do in each level of the recursion tree?


$T(n)$
$T(\frac{n}{2})$ $T(\frac{2n}{3})$ n^2
$T(\frac{1}{2}\frac{n}{2})$ $T(\frac{2}{3}\frac{n}{2})$ $\left(\frac{n}{2}\right)^2$ $T(\frac{1}{2}\frac{2n}{3})$ $T(\frac{2}{3}\frac{2n}{3})$ $\left(\frac{2n}{3}\right)^2$
$T(\frac{1}{2}\frac{n}{2})$ $T(\frac{2}{3}\frac{n}{2})$ $\left(\frac{n}{2}\right)^2$ $T(\frac{1}{2}\frac{2n}{3})$ $T(\frac{2}{3}\frac{2n}{3})$ $\left(\frac{2n}{3}\right)^2$ $T(\frac{1}{2}\frac{n}{2})$ $T(\frac{2}{3}\frac{n}{2})$ $\left(\frac{n}{2}\right)^2$ $T(\frac{1}{2}\frac{2n}{3})$ $T(\frac{2}{3}\frac{2n}{3})$ $\left(\frac{2n}{3}\right)^2$

In level 0, we take n^2 time. At level 1, the two subproblems take time:

$$\left(\frac{1}{2}n\right)^2 + \left(\frac{2}{3}n\right)^2 = \left(\frac{1}{4}+\frac{4}{9}\right)n^2 = \left(\frac{25}{36}\right)n^2$$

At level 2 the four subproblems are of size $\frac{1}{2}\frac{n}{2}$, $\frac{2}{3}\frac{n}{2}$, $\frac{1}{2}\frac{2n}{3}$, and $\frac{2}{3}\frac{2n}{3}$ respectively. These two subproblems take time:

$$\left(\frac{1}{4}n\right)^2 + \left(\frac{1}{3}n\right)^2 + \left(\frac{1}{3}\right)n^2 + \left(\frac{4}{9}\right)n^2 = \frac{625}{1296}n^2 = \left(\frac{25}{36}\right)^2 n^2$$

Similarly the amount of work at level k is at most $\left(\frac{25}{36}\right)^k n^2$.
Let $\alpha = \frac{25}{36}$, the total runtime is then:

$$\begin{aligned}
T(n) &\le \sum_{k=0}^{\infty} \alpha^k n^2 \\
&= \frac{1}{1-\alpha} n^2 \\
&= \frac{1}{1-\frac{25}{36}} n^2 \\
&= \frac{1}{\frac{11}{36}} n^2 \\
&= \frac{36}{11} n^2 \\
&= O(n^2)
\end{aligned}$$

That is, the first level provides a constant fraction of the total runtime.