COMPUTER SCIENCE & INFORMATION TECHNOLOGY

Algorithms

Comprehensive Theory
with Solved Examples *and* Practice Questions

MADE EASY PUBLICATIONS

www.madeeasypublications.org

MADE EASY Publications Pvt. Ltd.

Corporate Office: 44-A/4, Kalu Sarai (Near Hauz Khas Metro Station), New Delhi-110016 | **Ph. :** 9021300500

Email : infomep@madeeasy.in | **Web :** www.madeeasypublications.org

Algorithms

EDITIONS

First Edition : 2015
Second Edition : 2016
Third Edition : 2017
Fourth Edition : 2018
Fifth Edition : 2019
Sixth Edition : 2020
Seventh Edition : 2021
Eighth Edition : 2022
Ninth Edition : 2023
Tenth Edition : 2024
Eleventh Edition : 2025

Twelfth Edition : 2026

CONTENTS

Algorithms

CHAPTER 1

Asymptotic Analysis of Algorithms 3-30

1.1 Need for Performance Analysis 3
1.2 Worst, Average and Best Cases 4
1.3 Asymptotic Notations 5
1.4 Analysis of Loops 9
1.5 Comparisons of Functions 19
1.6 Asymptotic Behaviour of Polynomials 20
Student Assignments 23

CHAPTER 2

Recurrence Relations 31-54

2.1 Introduction 31
2.2 Substitution Method 32
2.3 Master Theorem 43
Student Assignments 46

CHAPTER 3

Divide and Conquer 55-89

3.1 Introduction 55
3.2 Quick Sort 55
3.3 Strassen's Matrix Multiplication 60
3.4 Merge Sort 63
3.5 Insertion Sort 66
3.6 Counting Inversions 67
3.7 Binary Search 69
3.8 Bubble Sort 72
3.9 Finding Min and Max 73
3.10 Power of An Element 76
Student Assignments 78

CHAPTER 4

Greedy Techniques 90-138

4.1 Introduction 90
4.2 Basic Examples of Greedy Techniques 91
4.3 Greedy Technique Formalization 92
4.4 Knapsack (Fractional) Problem 93
4.5 Representations of Graphs 96
4.6 Minimum Cost Spanning Tree (MCST) Problem 98
4.7 Single Source Shortest Path Problem (SSSPP) 107
4.8 Huffman Coding 117
Student Assignments 121

CHAPTER 5

Graph Based Algorithms 139-162

5.1 Introduction 139
5.2 Graph Searching 139
5.3 Directed Acyclic Graphs (DAG) 151
5.4 Topological Sorting 152
Student Assignments 155

CHAPTER 6

Dynamic Programming 163-195

6.1 Introduction 163
6.2 Fibonacci Numbers 163
6.3 All-Pairs Shortest Paths Problem 166
6.4 Matrix Chain Multiplication 170
6.5 The 0/1 Knapsack Problem 183
6.6 Multistage Graph 187
6.7 Traveling-Salesman Problem 189
Student Assignments 192

Algorithms

INTRODUCTION

In this book we tried to present the algorithms in a most simplified way. Each topic required for GATE is crisply covered with illustrative examples and each chapter is provided with Student Assignment at the end of each chapter so that the students get the thorough revision of the topics that he/she had studied. This subject is carefully divided into six chapters as described below:

Chapter 1: Asymptotic Analysis of Algorithms: In this chapter we discuss the asymptotic notations to represent the average, worst and best cases and we also learn how to identify time complexity of given algorithm.

Chapter 2: Recurrence Relations: In this chapter we study the three methods to study solve recurrence relations. The three methods are substitution method, master theorem and recursion tree method.

Chapter 3: Divide Conquer: In this chapter we discuss the various algorithms that can be implemented using divide and conquer paradigm namely merge and quick sort, Strassen's matrix multiplications. We also discuss the other sorting techniques.

Chapter 4: Greedy Algorithms: In this chapter we discuss the graph representations, algorithms' that require local optimizations at each step. Those algorithms include job scheduling, fractional Knapsack, Prim's and Kruskal's algorithms for MST's and other algorithms for shortest paths.

Chapter 5: Graph based Algorithms: In this chapter we discuss the BFS and DFS traversals. We also discuss the topological sorting algorithm.

Chapter 6: Dynamic Programming: In this chapter we discuss the algorithms for which greedy fails to give correct solution. We will discuss matrix chain multiplication, Floyd Warshall's algorithm, 0/1 Knapsack, longest common sequences and other algorithms.

■■■■

CHAPTER

1

Asymptotic Analysis of Algorithms

1.1 NEED FOR PERFORMANCE ANALYSIS

There are many important things that should be taken care of, like user friendliness, modularity, security, maintainability, etc. Why to worry about performance?

The answer to this is simple — we can have all the above things only if we have performance.

Goal: To write an algorithm which takes minimum time irrespective of the machine whether it is supercomputer or normal desktop.

Choosing the Best Algorithm

One naive way of doing this is – implement both the algorithms and run the two programs on your computer for different inputs and see which one takes less time. There are many problems with this approach for analysis of algorithms.

1. It might be possible that for some inputs, first algorithm performs better than the second. And for some inputs second performs better.
2. It might also be possible that for some inputs, first algorithm perform better on one machine and the second works better on other machine for some other inputs.

ASYMPTOTIC ANALYSIS is the big idea that handles above issues in analyzing algorithms. In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size (we don't measure the actual running time). We calculate, how does the time (or space) taken by an algorithm increases with the input size.

For example, let us consider the search problem (searching a given item) in a sorted array. One way to search is Linear Search (order of growth is linear) and other way is Binary Search (order of growth is logarithmic). To understand how Asymptotic Analysis solves the above mentioned problems in analyzing algorithms, let us say we run the Linear Search on a fast computer and Binary Search on a slow computer. For small values of input array size n, the fast computer may take less time. But, after certain value of input array size, the Binary Search will definitely start taking less time compared to the Linear Search even though the Binary Search is being run on a slow machine. The reason is the **order of growth** of Binary Search with respect to input size is logarithmic while the order of growth of Linear Search is linear. So the machine dependent constants can always be ignored after certain values of input size.

Asymptotic Analysis is not perfect, but that's the best way available for analyzing algorithms. For example, say there are two sorting algorithms that take 1000 *n* log *n* and 2 *n* log *n* time respectively on a machine. Both of these algorithms are asymptotically same (order of growth is *n* log *n*). So, With Asymptotic Analysis, we can't judge which one is better as we ignore constants in Asymptotic Analysis. Also, in asymptotic analysis, we always talk about input sizes larger than a constant value. It might be possible that those large inputs are never given to your software and an algorithm which is asymptotically slower always performs better for your particular situation. So, you may end up choosing an algorithm that is asymptotically slower but faster for your software.

Types of Asymptotic analysis:

1. Apriori analysis of algorithm. 2. Apostiari analysis of algorithms.

Apriori analysis	Apostiari analysis
1. It means we do analysis (space and time) of an algorithm prior to running it on specific system.	1. It means we do analysis of algorithm only after running it on system.
2. It does not depends on system.	2. It directly depends on system and changes from system to system
3. It provides approximate answer.	3. It gives exact answer.

1.2 WORST, AVERAGE AND BEST CASES

We can have three cases to analyze an algorithm:

1. Worst Case 2. Average Case 3. Best Case

Let us consider the following implementation of Linear Search.

```
#include <stdio.h>
// Linearly search x in arr []. If x is present then return the index,
// otherwise return -1
int search(int arr[], int n, int x)
{
      int i;
      for (i=0; i<n; i++)
      {
            if (arr[i] == x)
            return i;
      }
      return -1;
}
/* Driver program to test above functions*/
int main()
{
      int arr[] = {1, 10, 30, 15};
      int x = 30;
      int n = sizeof(arr)/sizeof(arr[0]);
      printf("%d is present at index %d", x, search(arr, n, x));
      getchar();
      return 0;
}
```

1.2.1 Worst Case Analysis

- In worst case analysis, we shall usually concentrate on finding only the **worst-case running time**, that is, the longest running time for any input of size n.
- The worst case analysis is an upper bound on the running time for any input. Knowing it gives us a guarantee that the algorithm will never take any longer.

Example: In the program given in section 1.2. That is for **linear search**, the worst case happens when the element to be searched is not present in the array (or) it is the last element, the search function compares it with all the elements of arr[ ] one by one. Therefore the worst case time complexity of linear search would be $\theta(n)$.

1.2.2 Average Case Analysis

- In average case analysis, we take all possible inputs and calculate computing time for all of the inputs.
- Considering all the cases, calculate the sum, and divide the sum by total number of inputs.

Example: For the linear search problem, let us assume that all cases are uniformly distributed (including the case of X not being present in array).

Considering all possible inputs:

i.e., when element found at 1^{st} position — $\theta(1)$
when element found at 2^{nd} position — $\theta(2)$
⋮
⋮
⋮
when element found at n^{th} position — $\theta(n)$

$$\text{Sum} = \sum_{i=1}^{n} \theta(i)$$

$$\text{Average case time} = \frac{\sum_{i=1}^{n} \theta(i)}{n} = \frac{\theta(n)(n+1)}{2} = \frac{\theta\left(\frac{n^2+n}{2}\right)}{n} = \theta(n)$$

1.2.3 Best Case Analysis

In the best case analysis, we calculate lower bound on running time of an algorithm. We must know the case that causes minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be $\Theta(1)$.

- Most of the times, we do worst case analysis to analyze algorithms. In the worst analysis, we guarantee an upper bound on the running time of an algorithm which is good information.
- The average case analysis is not easy to do in most of the practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs.
- The Best Case analysis is bogus. Guaranteeing a lower bound on an algorithm doesn't provide any information as in the worst case, an algorithm may take years to run.

1.3 ASYMPTOTIC NOTATIONS

Let f be a non negative function. Then we can define the three most common asymptotic bounds as follows.

1.3.1 Big-Oh(O)

We say that $f(n)$ is Big-O of $g(n)$, written as $f(n) = O(g(n))$, iff there are positive constants c and n_0 such that

$$0 \le f(n) \le c \cdot g(n) \text{ for all } n \ge n_0$$

If $f(n) = O(g(n))$, we say that $g(n)$ is an **upper bound** on $f(n)$.

Example 1.1 Let us consider a given function:

$f(n) = 4 \cdot n^3 + 10n^2 + 5n + 8$

$g(n) = n^3$

Checking whether $f(n) = O(g(n))$ or not?

Solution:

For above condition to be true

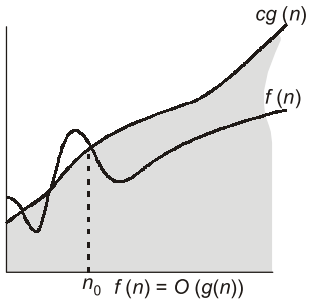
$0 \le f(n) \le c \cdot g(n)$

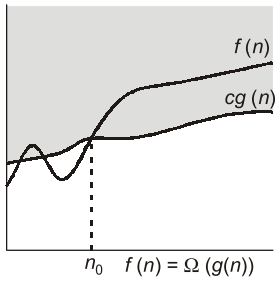
$4n^3 + 10n^2 + 5n + 8 \le c \cdot n^3$

when $c = 5$ and $n \ge 4$

$f(n)$ is always lesser than $g(n)$

Hence above statement is true.





1.3.2 Big-Omega (Ω)

We say that $f(n)$ is Big-Omega of $g(n)$, written as $f(n) = \Omega(g(n))$, iff there are positive constants c and n_0 such that

$$0 \le c \cdot g(n) \le f(n) \text{ for all } n \ge n_0$$

If $f(n) = \Omega(g(n))$, we say that $g(n)$ is a **lower bound** on $f(n)$.

Example 1.2 Let us consider a given function:

$f(n) = 3n + 2$

$g(n) = n$

Checking whether $f(n) = \Omega(g(n))$ or not?

Solution:

For above condition to be true

$0 \le c \cdot g(n) \le f(n)$

$c \cdot n \le 3n + 2$

when $c = 1$ and $n \geq 1$
$g(n)$ is always lesser than $f(n)$
Hence above statement is true.

1.3.3 Big-Theta (Θ)

We say that $f(n)$ is **Big-Theta** of $g(n)$, written as $f(n) = \Theta(g(n))$, iff there are positive constants c_1, c_2 and n_0 such that

$$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \text{ for all } n \geq n_o$$

Equivalently, $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. If $f(n) = \Theta(g(n))$, we say that $g(n)$ is a **tight bound** on $f(n)$.

Example 1.3 Consider a given function:

$f(n) = 3n^2 + 6n + 4$

$g(n) = n^2$

Checking whether $f(n) = \theta(g(n))$ or not?

Solution:

For above condition to be true, f and g must satisfy two conditions:

(i) $0 \leq f(n) \leq c_2 \cdot g(n)$

(ii) $0 \leq c_1 \cdot g(n) \leq f(n)$

Let's see whether there exists **some c_1 and c_2** or not.

(i) $3n^2 + 6n + 4 \leq c_2 \cdot n^2$

when $c_2 = 4, n \geq 3$

Above statement is true.

$\Rightarrow$ $F(n) = O(g(n))$

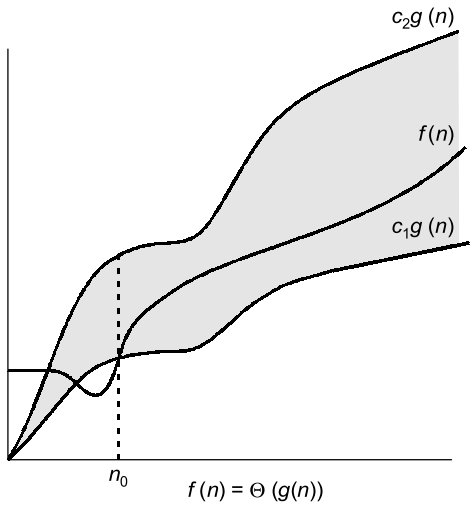
(ii) $c_1 \cdot n^2 \leq 3n^2 + 6n + 4$

when $c_1 = 2, n \geq 1$

Above statement is true.

$\Rightarrow$ $F(n) = \Omega(g(n))$

Hence, as above statement says $F(n) = \theta(g(n))$ if and only if $F(n) = O(g(n))$ and $F(n) = \Omega(g(n))$.



NOTE: Sometimes the notation $f(n) \le O(g(n))$ is used instead of $f(n) = O(g(n))$ (similar for Ω and Θ). These mean essentially the same thing, and the use of either is generally personal preference.

1.3.4 Small-Oh (o) Notation

The definitions of big-oh notation and small-oh notation are similar except one minor difference.

Big-oh	Small-oh
If $f(n) = o(g(n))$ The bound, $o \le f(n) \le \mathbf{c} \cdot (\mathbf{g(n)})$ holds for **some** constant $c > 0$ and all $n \ge n_0$	If $f(n) = o(g(n))$ The bound, $o \le \mathbf{f(n)} < c \cdot (\mathbf{g(n)})$ holds for **all** constant $c > 0$ and all $n > n_0$
Ex. $2n^2 = o(n^2)$ $\Rightarrow$ $2n^2 \le c \cdot n^2$ for $c = 3$ above condition true.	**Ex.** $2n^2 = o(n^2)$ $2n^2 < c \cdot n^2$ for $c = 3, 4, \ldots$ above condition is true but not for $c = 1$ hence, $2n^2 \ne o(n^2)$
Big-oh notation is used to denote an upper bound. **Ex.** (i) $2n = o(n)$ (ii) $3n^2 = o(n^2)$	Small-oh notation is used to denote an upper bound that is not asymptotically tight. **Ex.** (i) $2n = o(n^2)$ (ii) $3n^2 = o(2^n)$

The asymptotic upper bound provided by o-notation may not be asymptotically tight e.g. $2n^2 = O(n^2)$ is asymptotically tight but $2n = O(n^2)$ is not.

Hence we use o-notiation to devote an upper bound that is not asymptotically tight.

$o(g(n)) = \{f(n)\}$: for every positive constant $c > 0$

there exist a constant $n_0 > 0$ such that

$$0 \le f(n) < cg(n) \text{ for all } n \ge n_0$$

eg. $2n = \{f(n)\}$ but $2n^2 \ne o(n^2)$

1.3.5 Small-Omega (ω) Notation

ω-notation is used to denote a lower bound that is not asymptotically tight.

$\omega(g(n)) = \{f(n)\}$: for every positive constant $n_0 > 0$

there exist a constant $n_0 > 0$ such that

$$0 \le cg(n) < f(n) \text{ for all } n \ge n_0$$

$f(n) \in \omega(g(n))$ if and only if $g(n) \in o(f(n))$

eg. $\frac{n^2}{2} = \omega(n)$ but $\frac{n^2}{2} \ne \omega(n^2)$

Big-Omega (Ω)	Small-Omega (ω)
If $f(n) = \Omega(g(n))$ The bound, $O \le c \cdot (g(n)) \le f(n)$ holds for **some** $c > 0$ and all $n \ge n_0$	If $f(n) = \omega(g(n))$ The bound, $O \le c \cdot (g(n)) < f(n)$ holds for **all** $c > 0$ and all $n \ge n_0$
Ex. $3n^2 = \Omega(n^2)$ $c \cdot n^2 < 3n^2$ for $c = 1$ above condition true Hence, $3n^2 = \Omega(n^2)$ is true.	**Ex.** $3n^2 = \omega(n^2)$ $\Rightarrow$ $c \cdot n^2 < 3n^2$ for $c = 1, 2$ above condition is true but for $c \ge 3$, it becomes false hence, $3n^2 \ne \omega(n^2)$
Big-Omega is used to denote a lower bound. **Ex.** (i) $n^2 = \Omega(n^2)$ (ii) $2^n = \Omega(p^x)$ where p is any polynomial of degree x.	Small-Omega is used to denote a lower bound but not tight. **Ex.** (i) $n^2 = \omega(n)$ (ii) $2^n = \omega(p^x)$ where p is any polynomial of degree x.

1.4 ANALYSIS OF LOOPS

1. **O(1):** Time complexity of a function (or set of statements) is considered as $O(1)$ if it doesn't contain loop, recursion and call to any other non-constant time function.
   // set of non-recursive and non-loop statements

   For example swap( ) function has $O(1)$ time complexity. A loop or recursion that runs a constant number of times is also considered as $O(1)$. For example the following loop is $O(1)$.

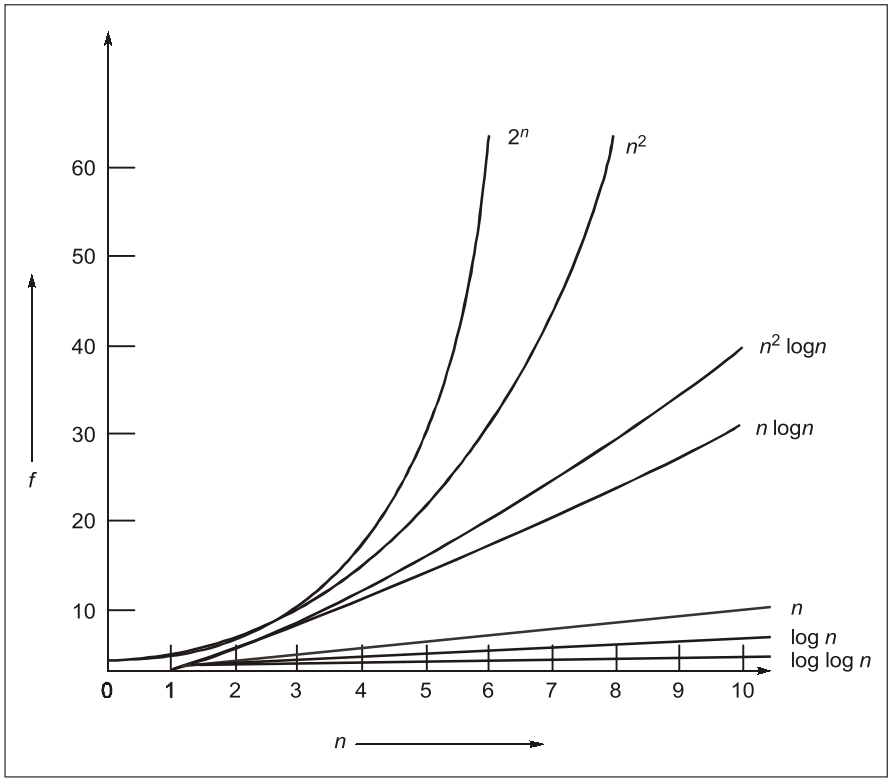
```
// Here c is a constant
for (int i = 1; i <= c; i++)
{
        // some O(1) expressions
}
```

2. **O(n):** Time Complexity of a loop is considered as $O(n)$ if the loop variables is incremented/decremented by a constant amount. For example following functions have $O(n)$ time complexity.

```
// Here c is a positive integer constant
for (int i = 1; i <= n; i += c)
{
        // some O(1) expressions
}
for (int i = n; i > 0; i -= c)
{
        // some O(1) expressions
}
```

3. **O(n^c):** Time complexity of nested loops is equal to the number of times the innermost statement is executed. For example the following sample loops have $O(n^2)$ time complexity.

```
for (int i = 1; i <= n; i += c)
{
        for (int j = 1; j <= n; j += c)
    {
```



$$O(1) < O(\log_2 n) < O(n) < O(n^2) < O(n^3) \ldots < O(n^k) < O(2^n)$$

Summary

- In the worst case analysis, we calculate upper bound on running time of an algorithm.
- In average case analysis, we take all possible inputs and calculate computing time for all of the inputs.
- In the best case analysis, we calculate lower bound on running time of an algorithm.
- We say that $f(n)$ is Big-O of $g(n)$, written as $f(n) = O(g(n))$
- We say that $f(n)$ is Big-Omega of $g(n)$, written as $f(n) = \Omega(g(n))$
- We say that $f(n)$ is **Big-Theta** of $g(n)$, written as $f(n) = \Theta(g(n))$
- $O(1) < O(\log_2 n) < O(n) < O(n^2) < O(n^3) \ldots < O(n^k) < O(2^n)$
- $\log(n!) < (\log(n))!$
- $\log(\log^* n) < \log^*(\log n)$
- $n < (\log n)^{\log n}$
- $2^n < n! < n^n$

Student's Assignments 1

Q.1 Which one of the following is true?

1. $an = o(n^2)\ a \geq 0$
2. $an^2 = O(n^2)\ a > 0$
3. $an^2 \neq o(n^2)\ a > 0$

(a) Only 1 and 2 are correct
(b) Only 1 is correct
(c) 1 and 3 are correct only
(d) All are correct

Q.2 Which of the following statements is true?

S1: $\frac{1}{2}n^2 = \omega(n)$ S2: $\frac{1}{2}n^2 = \omega(n^2)$

(a) S1 is correct
(b) S2 is correct
(c) S1 and S2 both are correct
(d) None of the above

Q.3 $f(n) = 3n^2 + 4n + 2$. Which will be the exact value for $f(n)$

(a) $\Theta(n^2)$ (b) $o(n^2)$
(c) $O(n^2)$ (d) $\Omega(n^2)$

Q.4 $f(n) = O(g(n))$ if and only if

(a) $g(n) = O(f(n))$ (b) $g(n) = \omega(f(n))$
(c) $g(n) = \Omega(f(n))$ (d) None of these

Q.5 $f(n) = o(g(n))$ if and only if

(a) $g(n) = \Omega(f(n))$
(b) $g(n) = \omega(f(n))$
(c) $g(n) = \Omega(f(n))$ and $g(n) = \omega(f(n))$
(d) None of these

Q.6 Which of the fallowing is not correct?

(a) $f(n) = O(f(n))$
(b) $c^* f(n) = O(f(n))$ for a constant C
(c) $(f(n) + g(n)) = o(g(n) + f(n))$
(d) None of the above

Q.7 $f(n) = \Theta(g(n))$ is

(a) $0 \leq C_1 g(n) \leq f(n) \leq C_2 g(n)\ \forall n \geq n_0$ where C_1, C_2, n_0 are + integer
(b) $0 \leq C_1 g(n) \leq f(n)\ \forall n \geq n_0$ where C_1, C_2, n_0 are + integer
(c) $0 \leq f(n) \leq C\ g(n)\ \forall n \geq n_0$ where C_1, n_0 are + integer
(d) None of the above

Q.8 $f(n) = O(g(n))$ implies

(a) $0 \leq C_1 g(n) \leq f(n)\ \forall n \geq n_0$ where C_1, n_0 are + integer
(b) $0 \leq C_1 f(n) \leq C_2 g(n)\ \forall n \geq n_0$ where C_1, C_2, n_0 are + integer
(c) $O \leq f(n) \leq C\ g(n)\ \forall n \geq n_0$ where C, n_0 are + integer
(d) $0 \leq C_1 g(n) \leq C_2 f(n)\ \forall n \geq n_0$ where C_1, C_2, n_0 are + integer

Q.9 $f(n) = \Theta(g(n))$ implies

(a) $f(n) = O(g(n))$ only
(b) $f(n) = \Omega(g(n))$ only
(c) $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
(d) None of the above

Q.10 $f(n) = o(g(n))$ implies

(a) $0 \leq f(n) \leq Cg(n)$ such that there exists some positive constant C and $n_0 > \forall n \geq n_0$
(b) $0 \leq f(n) < Cg(n)$ for every +ve constant $C >$ 0 there exists $n_0 > 0$, $\forall n \geq n_0$
(c) $0 \leq C_1 f(n) \leq C_2 g(n)\ \forall n \geq n_0$ such that C_1, C_2 and n_0 are +ve constants
(d) $0 \leq f(n) < C\ g(n)$ for some $+ve$ constant $C >$ $0\ \exists\ n_0 > 0$, $\forall n \geq n_0$

Q.11

```
void x (int A[ ]), int n)
{   int i, j;
    for (i = 0; i < n; i++)
    {   j = n – 1;
        while (j > i)
        {
            swap (A[j], A[j – 1]);
            j --;
        }
    }
}
```

What will be time complexity of the above algorithm if swap function takes constant time?

(a) $O(n)$ (b) $O(n^2)$
(c) $O(n \log_2 n)$ (d) $O(n^3)$

Q.12 Which of the following statements are True?

(a) $100\, n\log n = O(n\log n/100)$

(b) $\sqrt{\log n} = O(\log\log n)$

(c) If $0 < x < y$ then $n^x = O(x^y)$

(d) $2^n \neq O(n^c)$ where c is a constant and $c > 0$

Q.13 Which of the following statements (k, m are constants) are True?

(a) $(n + k)^m = O(n^m)$ (b) $2^{n+1} = O(2^n)$

(c) $2^{2^n} = O(2^n)$ (d) $f(n) = O(f(n)^2)$

Q.14 Which of the following statements are True?

(a) $n^2 \cdot 2^{3\log_2 n} = \Theta(n^5)$

(b) $\frac{4^n}{2^n} = \Theta(2^n)$

(c) $2^{\log_2 n} = \Theta(n^2)$

(d) if $f(n) = O(g(n))$ then $2^{f(n)} = O(2^{g(n)})$

Q.15 Consider $f(n)$, $g(n)$ and $h(n)$ be function defined as follows:

$$f(n) = \Omega(n^3)$$
$$g(n) = O(n^2)$$
$$h(n) = \Theta(n^2)$$

Which of the following represents correct asymptotic solution for $f(n) + [g(n) \times h(n)]$?

(a) $\Omega(n^3)$ (b) $O(n^4)$

(c) $\Theta(n^4)$ (d) $O(n^3)$

Q.16 Consider the following C-function:

```
int Rec (int n)
{
    int i, j, k, p, q = 0;
    for (i = n; i > 1; i/2) {
        p = 0;
        for (j = 1; j < n; j++)
            p = p +1;
        for (k = 1; k < p; k = k × 3)
            q++;
    } return q
}
```

Then time complexity of Rec in term of Θ notation is

(a) $\Theta(n)$ (b) $\Theta(n^2)$

(c) $\Theta(n \log n)$ (d) $\Theta(n \log\log n)$

Q.17 Consider following functions:

$$f(n) = (\log n)^{n-1}$$
$$g(n) = 2^n$$
$$h(n) = e^n / n$$

Which of the following is true?

(a) $f(n) = O(h(n))$ (b) $f(n) \neq O(n(n))$

(c) $g(n) = \Omega(f(n))$ (d) None of these

Q.18 Consider the following C function:

```
void dosomething (int n)
{
    int m, j, k;
    for ( j = 0; j < 200; j++)
    {
        for (k = 0; k < n; k++)
        {
            for (m = 0; m < j, m++)
            printf("%d", i + m);
        }
    }
}
```

What is the time complexity of the above function?

(a) $O(n^2)$ (b) $O(n\log n)$

(c) $O(n)$ (d) $O(n^2 \log n)$

Q.19 Let $g(n) = \Omega(n)$, $f(n) = O(n)$ and $h(n) = \theta(n)$ then what is the time complexity of $[g(n)\ f(n) + h(n)]$

(a) $O(n)$ (b) $\theta(n)$

(c) $\Omega(n)$ (d) $\theta(n^2)$

Q.20 Consider the following functions:

$$f_1 = n^4,\ f_2 = 4^n,\ f_3 = n^{110/37},\ f_4 = \left(\frac{119}{37}\right)^n$$

Which of the following is correct order of increasing growth rate?

(a) f_1, f_3, f_2, f_4 (b) f_3, f_1, f_4, f_2

(c) f_3, f_1, f_2, f_4 (d) f_1, f_3, f_4, f_2

Q.21 Consider the following program segments:

```
main ( )
    {
        int i = 0, m = 0;
    for (i = 1; i < n; i++) {
            for ( j = 1; j < i * i; j++) {
                if (( j % i) = = 0)
            for (k = 1; k < j; k++) {
```

```
                    m = m + 1; }
                }
            }
        }
```

What is the time complexity of above program?

(a) $O(n^2 \log n)$ (b) $O(n^3)$

(c) $O(n^4)$ (d) $O(n \log n)$

Answer Key:

1. (d)	**2.** (a)	**3.** (a)	**4.** (c)	**5.** (c)
6. (c)	**7.** (a)	**8.** (c)	**9.** (c)	**10.** (b)
11. (b)	**12.** (a, c, d)	**13.** (a, b)	**14.** (a, b)	**15.** (a)
16. (c)	**17.** (b)	**18.** (c)	**19.** (c)	**20.** (b)
21. (c)				

Student's Assignments Explanations

1. (d)

1. $an = o(n^2)$, $a \geq 0$
   $o \leq an < c \cdot n^2$ for all value of c this condition is true. Hence 1 is correct.
2. $an^2 = O(n^2)$
   $O \leq an^2 \leq c \cdot n^2$ when $c \geq a$ this condition is true. Hence, 2 is correct.
3. $an^2 \neq o(n^2)$
   for this statement to be false $o \leq an^2 < c \cdot n^2$ must be true

Since above condition needs to be true for all c thus if we could prove for some c that violates above condition. Whole option becomes true.

$O \leq an^2 < c \cdot n^2$ is not true when $c < a$ thus $an^2 \neq o(n^2)$ is true.

2. (a)

S1: $c \cdot \frac{1}{2} n^2 > n$ for every $c > 0$ which can be seen easily.

S2: False since $c \cdot \frac{1}{2} n^2 > n^2$ is false when $c = 2$ hence for every c it is not true thus, false.

3. (a)

$$f(n) = 3n^2 + 4n + 2$$

maximum degree = 2

$\Rightarrow \theta(n^2)$

Note: option (c) and (d) are also possible but they are not exact.

4. (c)

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

Take some arbitrary values in order to solve these type of questions.

As given, $f(n) = O(g(n))$

$\Rightarrow$ $f(n) \leq g(n)$

Let $f(n) = n$ and $g(n) = n^2$

(a) $g(n) = O(f(n)) \rightarrow n^2 = O(n)$ or $n^2 \leq n$ not true

(b) $g(n) = \omega(f(n)) \rightarrow g(n) > f(n) \rightarrow n^2 > n$
True (only for assumed value).
But it become false when both f and g are n.

(c) $g(n) = \Omega(f(n)) \rightarrow g(n) \geq f(n)$
Same thing written as given in question.

5. (c)

$$f(n) = o(g(n))$$

$\Rightarrow f(n)$ is strictly lesser than $g(n)$

Thus, $f(n) < g(n)$

For (c), $g(n) = \Omega(f(n))$

$\Rightarrow$ $g(n) \geq f(n)$

Satisfied by given equation.

$\Rightarrow$ $g(n) = \omega(f(n))$

$\Rightarrow$ $g(n) > f(n)$ this is also satisfied

6. (c)

(a) $f(n) = O(f(n))$ by reflexive property.

(b) $c * f(n) = O(f(n))$ since constant does not matter in case of asymptotic analysis.

(c) $(f(n) + g(n)) = o(g(n) + f(n))$

let $f(n) = n^2$

$g(n) = n$

$(n^2 + n) = o(n^2 + n)$

$n^2 = o(n^2)$ not true since tightest bound not allowed.

9. (c)

Go Through the Topic θ notation.

$F(n) = \theta(g(n))$ if and only if $f(n) = O(g(n))$ and $F(n) = \Omega(g(n)$ simultaneously.

10. (b)

As Small-oh definition says

$$f(n) = o(g(n))$$

when $O \le f(n) < c \cdot g(n)$ for every $c > 0$ and for all $n > n_0$

This is the major point.

Go Through the difference between Big-oh and Small-oh notation for this concept.

Option (d) is false since for some +ve constant is written there.

11. (b)

When $i = 0$

j loop i.e. while loop runs for $(n-1)$ time

when $i = 1$

j loop i.e. while loop runs for $(n-2)$ time

$\vdots$

when $i = n-1$

j loop will not run

Hence, $TC = 1 + 2 + \ldots (n-1)$

$$= \theta\left\{\frac{(n-1)(n)}{2}\right\}$$

i.e. $\theta(n^2)$

12. (a, c, d)

(a) $100\, n\log n = O(n\log n/100)$

constant does not matter. So it becomes

$$n\log n = O(n\log n)$$

Reflexive property. (True)

(b) $\sqrt{\log n} = O(\log\log n)$

$$\Rightarrow \quad \sqrt{\log n} \le \log\log n$$

Taking log on both sides

$$\frac{1}{2}\underbrace{\log\log n}_{x} \le \log(\log\log n)$$

$x \le \log x$ but this is not valid since $x > \log x$ thus, false statement.

(c) If $0 < x < y$ then $n^x = O(x^y)$

Exponential grows faster than polynomial.

Hence, true.

(d) $2^n \ne O(n^c)$ since $n^c = O(2^n)$ same reason as of c option.

13. (a, b)

(a) $(n+k)^m = O(n^m)$

Take $m = 2$ and $k = 1$

$$(n+1)^2 = O(n^2)$$

$$n^2 + 1 + 2n = O(n^2) \quad \text{(True)}$$

(b) $2^n \cdot 2 = O(2^n)$ true since constant does not matter.

(c) $2^{2^n} = O(2^n)$

Let, $2^n = x$

$2^x = O(x)$ (False)

Exponential grows faster.

(d) $f(n) = O(f(n)^2)$

Since we know $x > x^2$ when $x < 1$ thus,

Taking $f(n) = \frac{1}{n}$

$$\Rightarrow \quad \frac{1}{n} = O\left(\frac{1}{n^2}\right)$$

False since $\frac{1}{n} > \frac{1}{n^2}$

14. (a, b)

(a) $n^2 \cdot 2^{3\log_2 n} = \Theta(n^5)$

Note: $x^{\log_x n} \Leftrightarrow n$

$\log_2 n^x \Leftrightarrow x\log_x n$

Thus, $n^2 \cdot 2^{\log_2 n^3}$

$n^2 \cdot n^3$

$n^5 = \text{RHS}$ (True)

(b) $\frac{4^n}{2^n} = \left(\frac{4}{2}\right)^n = 2^n = \theta(2^n)$ (True)

(c) $2^{\log_2 n} = \theta(n^2)$

$n = \theta(n^2)$ not true since, $n \ne \Omega(n^2)$

(d) If $f(n) = O(g(n))$ then $2^{f(n)} = O(2^{g(n)})$

Let $f(n) = 2n$

$g(n) = n$

$\Rightarrow \quad 2n = O(n)$ (True)

But $2^{2n} = O(2^n)$ false since for this to be true c must be 2^n but c cannot be any function since it is constant.

15. (a)

- $f(n) = \Omega(n^3)$

$f(n) \ge C_1(n^3)$